VisualDSP++® 4.5
C/C++ Compiler and Library Manual for TigerSHARC® Processors

Revision 3.0, April 2006

Part Number
82-000336-03

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

©2006 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, the CROSSCORE logo, VisualDSP++, SHARC, TigerSHARC, and EZ-KIT Lite are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose of This Manual xxxvii
Intended Audience xxxvii
Manual Contents Description xxxviii
What's New in This Manual xxxviii
Technical or Customer Support xxxix
Supported Processors xl
Product Information xl
MyAnalog.com xl
Processor Product Information xli
Related Documents xlii
Online Technical Documentation xliii
Accessing Documentation From VisualDSP++ xliii
Accessing Documentation From Windows xliv
Accessing Documentation From the Web xliv
Printed Manuals xlv
VisualDSP++ Documentation Set xlv

Hardware Tools Manuals xlv
Processor Manuals xlv
Data Sheets xlv
Notation Conventions xlvii

COMPILER

C/C++ Compiler Overview 1-3
Compiler Command-Line Interface 1-5
Running the Compiler 1-6
Compiler Command-Line Switches 1-11
C/C++ Compiler Switch Summaries 1-11
C/C++ Mode Selection Switch Descriptions 1-22
-c89 1-22
-c++ 1-22
C/C++ Compiler Common Switch Descriptions 1-23
sourcefile 1-23
-@ filename 1-23
-A name [(<tokens>)] 1-23
-add-debug-libpaths 1-25
-align-branch-lines 1-25
-allow-macs-to-extend-saturation 1-25
-alttok 1-25
-always-inline 1-26
-annotate-loop-instr 1-27
-auto-attrs 1-27

-bss 1-27
-build-lib 1-27
-C 1-27
-c 1-28
-char-size-any 1-28
-char-size-{8|32} 1-28
-const-read-write 1-29
-const-strings 1-29
-Dmacro[=definition] 1-29
-debug-types 1-30
-default-branch-{np|p} 1-30
-double-size-any 1-31
-double-size-{32 | 64} 1-31
-dry 1-32
-dryrun 1-32
-E 1-33
-ED 1-33
-EE 1-33
-enum-is-int 1-34
-extra-keywords 1-34
-file-attr name[=value] 1-34
-flags-{asm|compiler|lib|link|mem} switch [,switch2 [,...]] 1-35
-force-circbuf 1-35
-fp-associative 1-35

-fp-div-lib 1-36
-full-version 1-36
-g 1-36
-glite 1-37
-H 1-37
-HH 1-37
-h[elp] 1-37
-I- 1-38
-I directory [{,|;} directory...] 1-38
-implicit-pointers 1-39
-include filename 1-40
-ipa 1-40
-L directory [{,|;} directory...] 1-40
-l library 1-40
-M 1-41
-MD 1-41
-MM 1-41
-Mo filename 1-42
-Mt name 1-42
-MQ 1-42
-map filename 1-42
-mem 1-42
-multiline 1-43
-never-inline 1-43

-no-align-branch-lines 1-43
-no-alttok 1-43
-no-annotate 1-43
-no-annotate-loop-instr 1-44
-no-auto-attrs 1-44
-no-bss 1-44
-no-builtin 1-45
-no-circbuf 1-45
-no-const-strings 1-45
-no-defs 1-45
-no-extra-keywords 1-46
-no-fp-associative 1-46
-no-fp-minmax 1-46
-no-mem 1-47
-no-multiline 1-47
-no-saturation 1-47
-no-std-ass 1-47
-no-std-def 1-47
-no-std-inc 1-48
-no-std-lib 1-48
-no-threads 1-48
-O 1-48
-O[0|1] 1-49
-Oa 1-49

-Og 1-49
-Os 1-50
-Ov num 1-50
-o filename 1-52
-overlay 1-52
-P 1-52
-PP 1-53
-path-{ asm | compiler | lib | link | mem } directory 1-53
-path-install directory 1-53
-path-output directory 1-53
-path-temp directory 1-53
-pch 1-54
-pchdir directory 1-54
-pedantic 1-54
-pedantic-errors 1-54
-pgo-session session-id 1-55
-pguide 1-56
-pplist filename 1-56
-proc processor 1-57
-progress-rep-func 1-57
-progress-rep-gen-opt 1-58
-progress-rep-mc-opt 1-58
-R directory [{:|,}directory …] 1-58
-R- 1-59

-S 1-59
-s 1-59
-save-temps 1-59
-section id=section_name[,id=section_name...] 1-60
-show 1-61
-si-revision version 1-61
-signed-bitfield 1-62
-signed-char 1-62
-structs-do-not-overlap 1-62
-syntax-only 1-63
-sysdefs 1-63
-T filename 1-64
-threads 1-64
-time 1-65
-Umacro 1-65
-unsigned-bitfield 1-65
-unsigned-char 1-66
-v 1-66
-verbose 1-66
-version 1-66
-W {error|remark|suppress|warn} number 1-67
-Werror-limit number 1-67
-Werror-warnings 1-67
-Wremarks 1-67

-Wterse 1-68
-w 1-68
-warn-protos 1-68
-workaround <workaround>[,<workaround>]* 1-68
-write-files 1-69
-write-opts 1-69
-xref <filename> 1-69
C++ Mode Compiler Switch Descriptions 1-71
-anach 1-71
-check-init-order 1-72
-eh 1-73
-full-dependency-inclusion 1-73
-ignore-std 1-75
-no-anach 1-75
-no-demangle 1-75
-no-eh 1-75
-no-implicit-inclusion 1-75
-no-rtti 1-75
-rtti 1-76
Data Types and Data Type Sizes 1-77
Integer Data Types 1-78
Floating-Point Data Types 1-78
Data Type Alignment 1-79
Environment Variables Used by the Compiler 1-80

Optimization Control 1-81
Interprocedural Analysis 1-84
Interaction with Libraries 1-85
C/C++ Compiler Language Extensions 1-87
Byte-Addressing Mode 1-91
sizeof() Operator Types and Sizes 1-91
Pointers 1-92
Alignment of Objects 1-92
Initializations 1-93
Pragmas Used in Byte-Addressing Mode 1-93
Performance Issues 1-93
Libraries Used in Byte-Addressing Mode 1-94
Include Files 1-95
Function Inlining 1-95
Inline Assembly Language Support Keyword (asm) 1-100
Assembly Construct Template 1-102
ASM() Construct Syntax: 1-102
ASM() Construct Syntax Rules 1-103
ASM() Construct Template Example 1-104
Assembly Construct Operand Description 1-105
Assembly Constructs With Multiple Instructions 1-110
Assembly Construct Reordering and Optimization 1-111
Assembly Constructs With Input and Output Operands .. 1-112
Assembly Constructs and Flow Control 1-113

Guidelines on the Use of asm() Statements 1-113
64-Bit Integer Support (long long) 1-114
Quad-Word Support 1-115
Memory Support Keywords (pm dm) 1-115
Memory Keyword Rules 1-115
__regclass Construct 1-117
Bank Type Qualifiers 1-118
Placement Support Keyword (section) 1-119
Placement of Compiler-Generated Code and Data 1-120
Boolean Type Support Keywords 1-121
Pointer Class Support Keyword (restrict) 1-121
Variable-Length Array Support 1-122
Non-Constant Aggregate Initializer Support 1-124
Indexed Initializer Support 1-124
Compiler Built-In Functions 1-126
Using the builtins.h Header File 1-127
Optimization Guidance Built-in Functions 1-129
16-Bit Data Types 1-131
Packed 16-bit Integer Support Using C 1-132
Constructors (int2x16 values) 1-132
Extractors and Expanders (int2x16 values) 1-133
Arithmetic Operators 1-133
Bitwise Operators (int2x16 values) 1-134
Comparison Operators (int2x16 values) 1-135

Sideways Sum (int2x16 values) 1-135
Constructors (int4x16 values) 1-136
Extractors (2x16 from a 4x16 value) 1-137
Arithmetic Operators (int4x16 values) 1-138
Sideways Sum (int4x16 values) 1-139
32-Bit Data Types 1-140
Constructors (int2x32 values) 1-140
Extractors (int2x32 values) 1-141
Circular Buffer Built-In Functions 1-142
Circular Buffer Increment of an Index 1-142
Circular Buffer Increment of a Pointer 1-142
Math Intrinsics 1-143
RECIPS 1-144
RSQRTS 1-145
Instructions Generated by Built-in Functions 1-145
Data Alignment Buffer (DAB) Built-in Functions 1-161
Circular Buffer Data Alignment Buffer (DAB) Built-in Functions 1-162
Communications Logic Unit Operations 1-164
TMAX, TMAX_ADD, TMAX_SUB, MAX_ADD, MAX_SUB 1-165
PERMUTE 1-170
ACS 1-171
DESPREAD 1-174
XCORRS 1-176

Pragmas 1-179
Data Alignment Pragmas 1-181
#pragma align num 1-182
#pragma alignment_region (alignopt) 1-183
Interrupt Handler Pragmas 1-185
Loop Optimization Pragmas 1-187
#pragma all_aligned 1-188
#pragma different_banks 1-188
#pragma loop_count(min, max, modulo) 1-188
#pragma loop_unroll N 1-189
#pragma no_alias 1-191
#pragma no_vectorization 1-192
#pragma vector_for 1-192
Function Side-Effect Pragmas 1-193
#pragma alloc 1-193
#pragma const 1-194
#pragma noreturn 1-194
#pragma pure 1-194
#pragma regs_clobbered string 1-195
#pragma regs_clobbered_call string 1-199
#pragma overlay 1-202
#pragma result_alignment (n) 1-203
General Optimization Pragmas 1-203
Inline Control Pragmas 1-204

#pragma always_inline 1-205
#pragma never_inline 1-206
Linking Control Pragmas 1-206
#pragma linkage_name identifier 1-206
#pragma core 1-207
#pragma section/#pragma default_section 1-212
#pragma file_attr (name[=value] [, name[=value] [...]]) 1-214
#pragma weak_entry 1-215
Class Conversion Optimization Pragmas 1-216
#pragma param_never_null param_name [...] 1-216
#pragma suppress_null_check 1-217
Template Instantiation Pragmas 1-219
#pragma instantiate instance 1-220
#pragma do_not_instantiate instance 1-220
#pragma can_instantiate instance 1-220
Header File Control Pragmas 1-221
#pragma hdrstop 1-221
#pragma no_implicit_inclusion 1-222
#pragma no_pch 1-223
#pragma once 1-223
#pragma system_header 1-223
Diagnostic Control Pragmas 1-223
Modifying the Severity of Specific Diagnostics 1-224
Modifying the Behavior of an Entire Class of Diagnostics 1-225

Saving or Restoring the Current Behavior of All Diagnostics .. 1-225
Memory Bank Pragmas 1-226
#pragma code_bank(bankname) 1-227
#pragma data_bank(bankname) 1-228
#pragma stack_bank(bankname) 1-229
#pragma bank_memory_kind(bankname, kind) 1-230
#pragma bank_read_cycles(bankname, cycles) 1-231
#pragma bank_write_cycles(bankname, cycles) 1-231
#pragma bank_optimal_width(bankname, width) 1-232
Increments and Decrements 1-233
C++ Style Comments 1-233
C++ Fractional Type Support 1-234
Format of Fractional Literals 1-234
Conversions Involving Fractional Values 1-235
Fractional Arithmetic Operations 1-235
Mixed-Mode Operations 1-236
GCC Compatibility Extensions 1-237
Statement Expressions 1-237
Type Reference Support Keyword (typeof) 1-238
GCC Generalized Lvalues 1-239
Conditional Expressions With Missing Operands 1-240
Hexadecimal Floating-Point Numbers 1-240
Zero-Length Arrays 1-241
Variable Argument Macros 1-241

Line Breaks in String Literals 1-242
Arithmetic on Pointers to Void and Pointers to Functions 1-242
Cast to Union 1-242
Ranges in Case Labels 1-242
Declarations Mixed With Code 1-243
Escape Character Constant 1-243
Alignment Inquiry Keyword (__alignof__) 1-243
(asm) Keyword for Specifying Names in Generated Assembler 1-244
Function, Variable and Type Attribute Keyword (__attribute__) 1-244
Unnamed struct/union fields within struct/unions 1-245
Preprocessor-Generated Warnings 1-245
Preprocessor Features 1-246
Predefined Preprocessor Macros 1-246
Writing Macros 1-249
C/C++ Run-Time Model and Environment 1-252
Stack Frame Overview 1-253
Stack Frame Description 1-255
General System-Wide Specifications 1-257
At a procedure call, the following must be true: 1-258
Argument Passage 1-259
Return Values 1-260
Procedure Call and Return 1-261
To Call a Procedure: 1-261
On Entry: 1-261

To Return from a Procedure: 1-262
Code Sequences 1-263
Support for argv/argc 1-264
Allocation of Memory for Stacks and Heaps in LDFs 1-265
Example of Heap/Stack Memory Allocation 1-265
Using Multiple Heaps 1-266
Heap Identifiers 1-267
Initializing the Heap 1-268
Using Alternate Heaps with the Standard Interface 1-268
Using the Alternate Heap Interface 1-269
C++ Run-time Support for the Alternate Heap Interface 1-270
Using the Heap_Install Interface 1-271
Miscellaneous Information 1-273
Register Classification 1-273
Callee Preserved Registers (“Preserved”) 1-273
Caller Save Registers (“Scratch”) 1-274
ADSP-TS101 and ADSP-TS20x Processor Registers 1-274
C/C++ and Assembly Language Interface 1-280
Calling Assembly Subroutines From C/C++ Programs 1-280
Calling C/C++ Functions From Assembly Programs 1-283
Using Mixed C/C++ and Assembly Naming Conventions 1-284
C++ Programming Examples 1-286
Using Fract Type Support 1-286
Using Complex Number Support 1-287

Compiler C++ Template Support 1-289
Template Instantiation 1-289
Identifying Un-instantiated Templates 1-291
File Attributes 1-293
Automatically-applied Attributes 1-293
Default LDF Placement 1-295
Sections versus Attributes 1-295
Granularity 1-296
“Hard” versus “soft” 1-296
Number of values 1-297
Using attributes 1-297
Example 1-298

ACHIEVING OPTIMAL PERFORMANCE FROM C/C++ SOURCE CODE

General Guidelines 2-3
How the Compiler Can Help 2-4
Using the Compiler Optimizer 2-4
Using Compiler Diagnostics 2-5
Warnings and Remarks 2-5
Source and Assembly Annotations 2-7
Using the Statistical Profiler 2-7
Using Profile-Guided Optimization 2-8
Using Profile-Guided Optimization With a Simulator 2-9

Using Profile-Guided Optimization With Non-simulatable Applications 2-10
Profile-Guided Optimization and Multiple Source Uses . 2-11
Profile-Guided Optimization and the -Ov switch 2-11
When to use Profile-Guided Optimization 2-12
Using Interprocedural Optimization 2-12
Data Types 2-13
Avoiding Emulated Arithmetic 2-14
Using Sub-Word Types with Caution 2-15
Getting the Most From IPA 2-16
Initialize Constants Statically 2-17
Quad-Word-Aligning Your Data 2-18
Using __builtin_aligned 2-18
Avoiding Aliases 2-20
Indexed Arrays Versus Pointers 2-22
Trying Pointer and Indexed Styles 2-23
Function Inlining 2-23
Using Inline asm Statements 2-24
Memory Usage 2-25
Putting Arrays into Different Memory Sections 2-25
Using the Bank Qualifier 2-28
Improving Conditional Code 2-28
Loop Guidelines 2-29
Keeping Loops Short 2-29
Avoiding Unrolling Loops 2-30

Avoiding Loop-Carried Dependencies 2-30
Avoiding Loop Rotation by Hand 2-31
Avoiding Array Writes in Loops 2-33
Inner Loops Versus Outer Loops 2-33
Avoiding Conditional Code in Loops 2-34
Avoiding Placing Function Calls in Loops 2-35
Avoiding Non-Unit Strides 2-35
Loop Control 2-36
Using the Restrict Qualifier 2-37
Using the Const Qualifier 2-38
Avoiding Long Latencies 2-38
Using Built-In Functions in Code Optimization 2-40
Using Fractional Data 2-40
System Support Built-in Functions 2-41
Using Circular Buffers 2-42
Smaller Applications: Optimizing for Code Size 2-44
Using Pragmas for Optimization 2-46
Function Pragmas 2-46
#pragma const 2-46
#pragma pure 2-47
#pragma regs_clobbered 2-47
#pragma optimize_{off|for_speed|for_space} 2-49
Loop Optimization Pragmas 2-49
#pragma loop_count 2-49

#pragma no_vectorization 2-50
#pragma vector_for 2-50
#pragma all_aligned 2-51
#pragma different_banks 2-53
#pragma no_alias 2-53
Useful Optimization Switches 2-54
How Loop Optimization Works 2-55
Terminology 2-55
Clobbered 2-55
Live 2-56
Spill 2-56
Scheduling 2-56
Loop kernel 2-56
Loop prolog 2-57
Loop epilog 2-57
Loop invariant 2-57
Hoisting 2-57
Sinking 2-58
Loop Optimization Concepts 2-58
Software pipelining 2-59
Loop Rotation 2-59
Loop Vectorization 2-62
Modulo Scheduling 2-64
Variable expansion 2-66

A Worked Example 2-69
Assembly Optimizer Annotations 2-72
Global Information 2-74
Procedure Statistics 2-75
Loop Identification 2-79
Loop Identification Annotations 2-80
File Position 2-84
Vectorization 2-85
Unroll and Jam 2-87
Loop Flattening 2-89
Vectorization Annotations 2-90
Modulo Scheduling 2-92
Warnings, Failure Messages and Advice 2-100

C/C++ RUN-TIME LIBRARY

C and C++ Run-Time Libraries Guide 3-3
Calling Library Functions 3-4
Using Compiler's Built-In C Library Functions 3-4
Linking Library Functions 3-5
Working With Library Source Code 3-8
Working With Library Header Files 3-9
assert.h 3-11
ctype.h 3-11
cycle_count.h 3-11
cycles.h 3-12

device.h 3-12
device_int.h 3-12
errno.h 3-13
float.h 3-13
iso646.h 3-14
limits.h 3-14
locale.h 3-15
math.h 3-15
setjmp.h 3-16
signal.h 3-16
stdarg.h 3-17
stddef.h 3-17
stdio.h 3-17
stdlib.h 3-20
string.h 3-20
time.h 3-21
DSP Header Files 3-22
complex.h – Basic Complex Arithmetic Functions 3-23
filter.h – DSP Filters and Transformations 3-24
libsim.h – Simulator Services 3-25
matrix.h – Matrix Functions 3-26
stats.h – Statistical Functions 3-27
vector.h – Vector Functions 3-28
window.h – Window Generators 3-29

Calling Library Functions from an ISR 3-31
Abridged C++ Library Support 3-32
Embedded C++ Library Header Files 3-32
complex 3-32
exception 3-33
fract 3-33
fstream 3-33
iomanip 3-33
ios 3-33
iosfwd 3-34
iostream 3-34
istream 3-34
new 3-34
ostream 3-34
sstream 3-34
stdexcept 3-34
streambuf 3-35
string 3-35
strstream 3-35
C++ Header Files for C Library Facilities 3-35
Embedded Standard Template Library Header Files 3-36
algorithm 3-36
deque 3-37
functional 3-37

hash_map 3-37
hash_set 3-37
iterator 3-37
list 3-37
map 3-37
memory 3-37
numeric 3-37
queue 3-38
set 3-38
stack 3-38
utility 3-38
vector 3-38
fstream.h 3-38
iomanip.h 3-38
iostream.h 3-38
new.h 3-39
Using the Thread-Safe C/C++ Run-Time Libraries with VDK 3-39
Measuring Cycle Counts 3-39
Basic Cycle Counting Facility 3-40
Cycle Counting Facility with Statistics 3-41
Using time.h to Measure Cycle Counts 3-44
Determining the Processor Clock Rate 3-46
Considerations When Measuring Cycle Counts 3-47
File I/O Support 3-49

Extending I/O Support To New Devices 3-50
DevEntry Structure 3-50
Registering New Devices 3-55
Pre-Registering Devices 3-56
Default Device 3-57
Remove and Rename Functions 3-58
Default Device Driver Interface 3-58
Data Packing For Primitive I/O 3-60
Data Structure for Primitive I/O 3-60
Documented Library Functions 3-65
Run-Time Library Reference 3-69
a_compress 3-70
a_expand 3-71
abs 3-72
acos 3-73
addbitrev 3-74
alog 3-75
alog10 3-77
arg 3-79
asctime 3-80
asin 3-82
atan 3-83
atan2 3-84
atof 3-85

atoi 3-87
atol 3-88
atold 3-89
atoll 3-91
autocoh 3-92
autocorr 3-93
avg 3-94
bsearch 3-95
cabs 3-97
cadd 3-98
cartesian 3-99
cdiv 3-101
ceil 3-102
cexp 3-103
cfft 3-104
cfft_mag 3-107
cfft2d 3-109
cfftf 3-111
clearerr 3-113
clip 3-114
clock 3-115
cmatmadd 3-117
cmatmmlt 3-119
cmatmsub 3-121

cmatsadd 3-123
cmatsmlt 3-124
cmatssub 3-126
cmlt 3-128
conj 3-129
convolve 3-130
conv2d 3-132
copysign 3-134
cos 3-135
cosh 3-136
cot 3-137
count_ones 3-138
crosscoh 3-139
crosscorr 3-140
csub 3-141
ctime 3-142
cvecdot 3-143
cvecsadd 3-144
cvecsmlt 3-145
cvecssub 3-146
cvecvadd 3-147
cvecvmlt 3-148
cvecvsub 3-149
difftime 3-150

div 3-152
exp 3-153
__emuclk 3-154
fabs 3-155
favg 3-156
fclip 3-157
fclose 3-158
feof 3-159
ferror 3-160
fir 3-161
fir_decima 3-163
fir_interp 3-166
fflush 3-172
fgetc 3-173
fgetpos 3-174
fgets 3-176
floor 3-178
fmax 3-179
fmin 3-180
fmod 3-181
fopen 3-182
fprintf 3-184
fputc 3-190
fputs 3-191

fread 3-192
freopen 3-194
frexp 3-196
fscanf 3-197
fseek 3-201
fsetpos 3-203
ftell 3-204
fwrite 3-205
gen_bartlett 3-207
gen_blackman 3-209
gen_gaussian 3-210
gen_hamming 3-211
gen_hanning 3-212
gen_harris 3-213
gen_kaiser 3-214
gen_rectangular 3-216
gen_triangle 3-217
gen_vonhann 3-219
getc 3-220
getchar 3-221
gets 3-222
gmtime 3-224
histogramf 3-226
ifft 3-228

ifft2d 3-230
iir 3-232
interrupt, interruptf, interrupts, interruptnr, interruptfnr, interruptsnr 3-235
localtime 3-240
log 3-242
log10 3-243
matinv 3-244
matmadd 3-245
matmmlt 3-246
matmsub 3-248
matsadd 3-249
matsmlt 3-250
matssub 3-251
max 3-252
mean 3-253
min 3-254
mktime 3-255
modf 3-257
mu_compress 3-258
mu_expand 3-259
norm 3-260
perror 3-261
polar 3-262
pow 3-263

printf 3-264
putc 3-265
putchar 3-266
puts 3-267
qsort 3-268
raise 3-270
rand 3-273
remove 3-274
rename 3-275
rewind 3-277
rfft 3-278
rfft_mag 3-281
rfft2d 3-283
rfftf 3-285
rfftf_mag 3-287
rms 3-289
rsqrt 3-290
scanf 3-291
setbuf 3-293
setvbuf 3-295
sign 3-297
signal, signalf, signals, signalnr, signalfnr, signalsnr 3-298
sin 3-302
sinh 3-303

snprintf 3-304
sprintf 3-306
sqrt 3-308
srand 3-309
sscanf 3-310
strftime 3-312
strtod 3-316
strtof 3-318
strtoi 3-320
strtol 3-322
strtold 3-324
strtoll 3-326
strtoul 3-328
strtoull 3-330
tan 3-332
tanh 3-333
time 3-334
transpm 3-335
twidfft 3-336
twidfftf 3-338
ungetc 3-340
var 3-342
vecdot 3-343
vecsadd 3-344

vecsmlt 3-345
vecssub 3-346
vecvadd 3-347
vecvmlt 3-348
vecvsub 3-349
vfprintf 3-350
vprintf 3-352
vsnprintf 3-354
vsprintf 3-356
zero_cross 3-358

INDEX

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for digital signal processing (DSP) applications.

Purpose of This Manual

The *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for TigerSHARC Processors* contains information about the C/C++ compiler and run-time library for TigerSHARC® (ADSP-TSxxx) processors. It includes syntax for command lines, switches, and language extensions. It leads you through the process of using library routines and writing mixed C/C++/assembly code.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe your target architecture.

Manual Contents Description

This manual contains:

- Chapter 1, "Compiler"
  Provides information on compiler options, language extensions and C/C++/assembly interfacing
- Chapter 2, "Achieving Optimal Performance from C/C++ Source Code"
  Provides information on compiler (and assembly) code optimization (techniques and options).
- Chapter 3, "C/C++ Run-Time Library"
  Shows how to use library functions and provides a complete C/C++ library function reference (for functions covered in the current compiler release)

What's New in This Manual

This edition of the *VisualDSP++ 4.5 C/C++Compiler and Library Manual for TigerSHARC Processors* documents support for all currently available TigerSHARC processors listed in "Supported Processors".

Refer to the *VisualDSP++ 4.5 Product Release Bulletin* for a complete list of new compiler features and enhancements.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at
  http://www.analog.com/processors/technicalSupport
- E-mail tools questions to
  processor.tools.support@analog.com
- E-mail processor questions to
  processor.support@analog.com (World wide support)
  processor.europe@analog.com (Europe support)
  processor.china@analog.com (China support)
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

Supported Processors

The name "TigerSHARC" refers to a family of Analog Devices, Inc. floating-point and [8-bit, 16-bit, and 32-bit] fixed-point processors. VisualDSP++ currently supports the following TigerSHARC processors:

ADSP-TS101	ADSP-TS202
ADSP-TS201	ADSP-TS203

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

`MyAnalog.com` is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests. `MyAnalog.com` provides access to books, application notes, data sheets, code examples, and more.

Registration

Visit `www.myanalog.com` to sign up. Click **Register** to use `MyAnalog.com`. Registration takes about five minutes and serves as a means to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at `www.analog.com/processors`, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)
- Access the FTP Web site at
  `ftp ftp.analog.com` (or `ftp 137.71.25.69`)
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see the following publications:

- *VisualDSP++ 4.5 Getting Started Guide*
- *VisualDSP++ 4.5 User's Guide*
- *VisualDSP++ 4.5 Assembler and Preprocessor Manual*
- *VisualDSP++ 4.5 Linker and Utilities Manual*
- *VisualDSP++ 4.5 Loader Manual*
- *VisualDSP++ 4.5 Product Release Bulletin*
- *VisualDSP++ Kernel (VDK) User's Guide*
- *Quick Installation Reference Card*

For hardware information, refer to your processors's hardware reference, programming reference, or data sheet. All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

```
http://www.analog.com/processors/resources/technical Library
```

Online Technical Documentation

Online documentation includes the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest using the Search function of VisualDSP++ Help system. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
`.CHM`	Help system files and manuals in Help format
`.HTM` or `.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the `.HTML` files requires a browser, such as Internet Explorer 5.01 (or higher).
`.PDF`	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the `.PDF` files requires a PDF reader, such as Adobe Acrobat Reader (4.5 or higher).

Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (.CHM) are located in the Help folder of VisualDSP++ environment. The .PDF files are located in the Docs folder of your VisualDSP++ installation CD-ROM. The Docs folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the vdsp-help.chm file, which is the master Help system, to access all the other .CHM files.
- Open your VisualDSP++ installation CD-ROM and double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.

Accessing Documentation From the Web

Download manuals in PDF format at the following Web site:
http://www.analog.com/processors/resources/technicalLibrary/manuals

Select a processor family and book title. Download archive (.ZIP) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto `http://www.analog.com/salesdir/continent.asp`.

Hardware Tools Manuals

To purchase EZ-KIT Lite® and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

Text conventions used in this manual are identified and described as follows.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

Additional conventions, which apply only to specific chapters, may appear throughout this document.

1 COMPILER

The C/C++ compiler (`ccts`) is part of Analog Devices development software for TigerSHARC (ADSP-TSxxx) processors.

The code examples in this manual have been compiled using VisualDSP++ 4.5. The examples compiled with other versions of VisualDSP++ may result in build errors or different output although the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

This chapter contains:

- "C/C++ Compiler Overview" on page 1-3
  provides an overview of C/C++ compiler for TigerSHARC processors.
- "Compiler Command-Line Interface" on page 1-5
  describes the operation of the compiler as it processes programs, including input and output files, and command-line switches.
- "C/C++ Compiler Language Extensions" on page 1-87
  describes the `ccts` compiler's extensions to the ISO/ANSI standard for the C and C++ languages.
- "Preprocessor Features" on page 1-246
  contains information on the preprocessor and ways to modify source compilation.
- "C/C++ Run-Time Model and Environment" on page 1-252
  contains reference information about implementation of C/C++ programs, data, and function calls in TigerSHARC processors

- "C/C++ and Assembly Language Interface" on page 1-280 describes how to call an assembly language subroutine from C or C++ program, and how to call a C or C++ function from within an assembly language program.
- "Compiler C++ Template Support" on page 1-289 describes how templates are instantiated at compile time

C/C++ Compiler Overview

The C/C++ compiler (`ccts`) is designed to aid your DSP project development efforts by:

- Processing C and C++ source files, producing machine-level versions of the source code and object files
- Providing relocatable code and debugging information within the object files
- Providing relocatable data and program memory segments for placement by the linker in the processors' memory

Using C/C++, developers can significantly decrease time-to-market since it gives them the ability to efficiently work with complex signal processing data types. It also allows them to take advantage of specialized signal processing operations without having to understand the underlying processor architecture.

The C/C++ compiler (`ccts`) compiles ISO/ANSI standard C and C++ code for the TigerSHARC processors. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in project development. The `ccts` compiler runs from the VisualDSP++ environment or from an operating system command line.

The C/C++ compiler processes your C and C++ language source files and produces TigerSHARC assembler source files. The assembler source files are assembled by the TigerSHARC assembler (`easmts`). The assembler creates Executable and Linkable Format (ELF) object files that can either be linked (using the linker) to create a executable file or included in an archive library (using `elfar`). The way in which the compiler controls the assemble, link, and archive phases of the process depends on the source input files and the compiler options used.

Source files contain the C/C++ program to be processed by the compiler. The `ccts` compiler supports the ANSI/ISO standard definitions of the C and C++ languages. For information on the C language standard, see any of the many reference texts on the C language. Analog Devices recommends the Bjarne Stroustrup text "*The C++ Programming Language*" from Addison Wesley Longman Publishing Co (ISBN: 0201889544) (1997) as a reference text for the C++ programming language.

The `ccts` compiler supports a set of C/C++ language extensions. These extensions support hardware features of the TigerSHARC processors. For more information, see "C/C++ Compiler Language Extensions" on page 1-87.

Compiler options are set in the VisualDSP++ Integrated Development and Debug Environment (IDDE) from the **Compile** page of the **Project Options** dialog box. The selections control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output, etc.

By default, the `ccts` compiler operates in the 32-bit word-addressing mode. The `ccts` compiler can also be set for the 8-bit byte-addressing mode. For more information, refer to "-char-size-any" switch (on page 1-28), "-char-size-{8|32}" switch (on page 1-28), "Data Types and Data Type Sizes" on page 1-77, and "Byte-Addressing Mode" on page 1-91.

For more information on the VisualDSP++ environment, see online Help.

Compiler Command-Line Interface

This section describes how the `ccts` compiler is invoked from the command line, the various types of files used by and generated from the compiler, and the switches used to tailor the compiler's operation.

This section contains:

- "Running the Compiler" on page 1-6
- "Compiler Command-Line Switches" on page 1-11
- "Data Types and Data Type Sizes" on page 1-77
- "Data Type Alignment" on page 1-79
- "Environment Variables Used by the Compiler" on page 1-80
- "Optimization Control" on page 1-81

By default, the compiler runs with Analog Extensions for C code enabled. This means that the compiler processes source files written in ANSI/ISO standard C language supplemented with Analog Devices extensions. Table 1-2 on page 1-8 lists valid extensions of source files the compiler operates upon. By default, the compiler processes the input file through the listed stages to produce a `.DXE` file. (See file names in Table 1-3 on page 1-9.) Table 1-4 on page 1-11 lists the switches that select the language dialect.

Although many switches are generic between C and C++, some of them are valid in C++ mode only. A summary of the generic C/C++ compiler switches appears in Table 1-5 on page 1-12. A summary of the C++ specific compiler switches appears in Table 1-6 on page 1-21. The summaries are followed by descriptions of each switch.

When developing a DSP project, sometimes it is useful to modify the compiler's default options settings. The way the compiler's options are set depends on the environment used to run the DSP development software.

Running the Compiler

Use the following general syntax for the `ccts` command line:

```
ccts [-switch [-switch …]] sourcefile [sourcefile …]
```

runs the assembler with

Table 1-1. TigerSHARC Command Line Syntax

Command Element	Description
`ccts`	Name of the compiler program for TigerSHARC processors.
`-switch`	Switch (or switches) to process. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case sensitive. For example, `-O` is not the same as `-o`.
`sourceFile`	Name of the file to be preprocessed, compiled, assembled, and/or linked

A file name can include the drive, directory, file name, and file extension. The compiler supports both Win32- and POSIX-style paths, using either forward or back slashes as the directory delimiter. It also supports UNC path names (starting with two slashes and a network name).

When file names or other switches for the compiler include spaces or other special characters, you must ensure that these are properly quoted (usually using double-quote characters), to ensure that they are not interpreted by the operating system before being passed to the compiler.

The `ccts` compiler uses the file extension to determine what the file contains and what operations to perform upon it. Table 1-3 on page 1-9 lists the allowed extensions.

For example, the following command line

```
ccts -proc ADSP-TS101 -O -Wremarks -o program.dxe source.c
```

runs `ccts` with

`-proc ADSP-TS101`	Specifies the processor
`-O`	Specifies optimization for the compiler
`-Wremarks`	Selects extra diagnostic remarks in addition to warning and error messages
`-o program.dxe`	Selects a name for the compiled, linked output
`source.c`	Specifies the C language source file to be compiled

The following example command line for the C++ mode

```
ccts -proc ADSP-TS101 -c++ source.cpp
```

runs `ccts` with:

`-c++`	Specifies that all of the source files to be compiled in C++
`source.cpp`	Specifies the C++ language source file for your program

The normal function of `ccts` is to invoke the compiler, assembler, and linker as required to produce an executable object file. The precise operation is determined by the extensions of the input filenames, and by various switches.

In normal operation the compiler uses the following extension files to perform a specified action:

If multiple files are specified, each is first processed to produce an object file; then all object files are presented to the linker.

Table 1-2. File Extensions

Extension	Action
`.c .cpp .cxx .cc .c++`	Source file is compiled, assembled, and linked
`.asm`, `.dsp`, or `.s`	Assembly language source file is assembled and linked
`.doj`	Object file (from previous assembly) is linked

You can stop this sequence at various points by using appropriate compiler switches, or by selecting options with the VisualDSP++ environment. These switches are `-E`, `-P`,`-M`,`-H`,`-S`, and `-c`.

Many of the compiler's switches take a file name as an optional parameter. If you do not use the optional output name switch, `ccts` names the output for you. Table 1-3 on page 1-9 lists the type of files, names, and extensions `ccts` appends to output files.

File extensions vary by command-line switch and file type. These extensions are influenced by the program that is processing the file, any search directories that you select, and any path information that you include in the file name. Table 1-3 indicates the searches that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file search paths. For information on additional search directories, see the `-I directory` switch (on page 1-38) and `-L directory` switch (on page 1-40).

When providing an input or output file name as an optional parameter, use the following guidelines:

- Use a file name (include the file extension) with either an unambiguous relative path or an absolute path. A file name with an absolute path includes the drive, directory, file name, and file extension.
- Enclose long file names within straight quotes; for example, `"long file name.c"`. The `ccts` compiler uses the file extension conventions listed in Table 1-3 to determine the input file type.

- Verify that the compiler is using the correct file. If you do not provide the complete file path as part of the parameter or add additional search directories, `ccts` looks for input in the current directory.

Using the verbose output switches for the preprocessor, compiler, assembler, and linker causes each of these tools to display command-line information as they process each file.

Table 1-3. Input and Output Files

Input File Extension	File Extension Description
`.c`	C/C++ source file
`.cc .cpp .cxx`	C++ source file
`.h`	Header file (referenced by a `#include` statement)
`.hpp .hh .hxx .h++`	C++ header file (referenced by a `#include` statement)
`.pch`	C++ pre-compiled header file
`.i`	Preprocessed C/C++ source, created when preprocess only (`-E` compiler switch) is specified
`.ipa, .opa`	Interprocedural analysis files—used internally by the compiler when performing interprocedural analysis
`.pgo`	Execution profile generated by a simulation run. For more information, see "Using Profile-Guided Optimization" in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.
`.s, .dsp, .asm`	Assembler source file
`.ii`	Template instantiation files—used internally by the compiler when instantiating C++ templates
`.is`	Preprocessed assembly source (retained when `-save-temps` is specified)
`.ldf`	Linker Description File
`.doj`	Object file to be linked
`.dlb`	Library of object files to be linked as needed

Table 1-3. Input and Output Files (Cont'd)

Input File Extension	File Extension Description
.xml	Processor system memory map file output
.sym	Processor system symbol map file output

The compiler refers to a number of environment variables during its operation, and these environment variables can affect the compiler's behavior. Refer to "Environment Variables Used by the Compiler" on page 1-80 for more information.

Compiler Command-Line Switches

This section describes the command-line switches used when compiling. It contains a set of tables that provide a brief description of each switch. These tables are organized by type of a switch. Following these tables are sections that provide fuller descriptions of each switch.

C/C++ Compiler Switch Summaries

This section contains a set of tables that summarize generic and specific switches (options), as follows:

- Table 1-4 "C or C++ Mode Selection Switches"
- Table 1-5 "C/C++ Compiler Common Switches" on page 1-12
- Table 1-6 "C++ Mode Compiler Switches" on page 1-21

A brief description of each switch follows the tables, beginning on page 1-22.

Table 1-4. C or C++ Mode Selection Switches

Switch Name	Description
`-c89` (on page 1-22)	Supports programs that conform to the ISO/IEC 9899:1990 standard
`-c++` (on page 1-22)	Supports ANSI/ISO standard C++ with Analog Devices extensions

Table 1-5. C/C++ Compiler Common Switches

Switch Name	Description
`-allow-macs-to-extend-saturation` (on page 1-25)	Instructs the compiler to try to generate multiply-accumulate instructions using saturating add and subtract operations.
`sourcefile` (on page 1-23)	Specifies file to be compiled
`-@ filename` (on page 1-23)	Reads command-line input from the file
`-A name[tokens]` (on page 1-23)	Asserts the specified name as a predicate
`-add-debug-libpaths` (on page 1-25)	Link against debug-specific variants of system libraries, where available.
`-align-branch-lines` (on page 1-25)	Quad align predicted branches
`-alttok` (on page 1-25)	Allows alternative keywords and sequences in sources
`-always-inline` (on page 1-26)	Treats `inline` keyword as a requirement rather than a suggestion.
`-annotate-loop-instr` (on page 1-27)	Provides additional annotation information for the prolog, kernel and epilog of a loop
`-auto-attrs` (on page 1-35)	Directs the compiler to emit automatic attributes based on the files it compiles. Enabled by default.
`-bss` (on page 1-27)	Causes the compiler to place global zero-initialized data into a separate BSS-style section
`-build-lib` (on page 1-27)	Directs the librarian to build a library file
`-C` (on page 1-27)	Retains preprocessor comments in the output file; active only with the `-E` or `-P` switch)
`-c` (on page 1-28)	Compiles and/or assembles only; does not link
`-char-size-any` (on page 1-28	Indicate that the resulting object can link against any `char` size object

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-char-size-{8|32}` (on page 1-28)	Selects byte (8-bit) or word (32-bit) addressing mode
`-const-read-write` (on page 1-29)	Constant pointers may access modifiable memory
`-D`*`macro`*`[=`*`definition`*`]` (on page 1-29)	Defines a macro
`-debug-types` (on page 1-30)	Supports building a `*.h` file directly and writing a complete set of debugging information for the header file
`-default-branch-{np|p}` (on page 1-30)	Sets default branches to be predict or non-predict
`-double-size-any` (on page 1-31)	Indicate that the resulting object can link against any `double` size object
`-double-size-{32|64}` (on page 1-31)	Selects 32- or 64-bit IEEE format for `double`; the `-double-size-32` is the default mode
`-dry` (on page 1-32)	Displays, but does not perform, main driver actions (verbose dry-run)
`-dryrun` (on page 1-32)	Displays, but does not perform, top-level driver actions (terse dry-run)
`-E` (on page 1-33)	Preprocesses, but does not compile, the source file
`-ED` (on page 1-33)	Preprocesses and sends all output to a file
`-EE` (on page 1-33)	Preprocesses and compiles the source file
`-enum-is-int` (on page 1-34)	By default `enums` can have a type larger than `int`. This option ensures the `enum` type is `int`.
`-extra-keywords` (on page 1-34)	Recognizes Analog Devices extensions to ISO/ANSI standards for C and C++ (default mode)
`-file-attr` *`name`*`[=`*`value`*`]` (on page 1-35)	Adds the specified attribute name/value pair to the file(s) being compiled

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-flags-{tools} <arg1> [,arg2...]` (on page 1-35)	Passes command-line switches through the compiler to other build tools
`-force-circbuf` (on page 1-35)	Treats array references of the form `array[i%n]` as circular buffer operations
`-fp-associative` (on page 1-35)	Treats floating-point multiply and addition as an associative
`-fp-div-lib` (on page 1-36)	Do not use inline code for floating-point divides
`-full-version` (on page 1-36)	Displays version information for build tools
`-g` (on page 1-36)	Generates DWARF-2 debug information
`-glite` (on page 1-37)	Generates lightweightDWARF-2 debug information
`-H` (on page 1-37)	Outputs a list of header files, but does not compile the source file
`-HH` (on page 1-37)	Outputs a list of included header files and compiles
`-h[elp]` (on page 1-37)	Outputs a list of command-line switches with brief syntax descriptions
`-I-` (on page 1-38)	Establishes the point in the `include` directory list at which the search for header files enclosed in angle brackets should begin
`-I` *`directory`* (on page 1-38)	Appends *directory* to the standard search path.
`-implicit-pointers` (on page 1-39)	Demotes incompatible-pointer-type errors into discretionary warnings. Not valid when compiling in C++ mode.
`-include` *`filename`* (on page 1-40)	Includes named file prior to preprocessing each source file

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-ipa` (on page 1-40)	Enables interprocedural analysis
`-L` *`directory`* (on page 1-40)	Appends the specified directory to the standard library search path when linking
`-l` *`library`* (on page 1-40)	Searches the specified library for functions when linking
`-M` (on page 1-41)	Generates make rules only; does not compile
`-MD` (on page 1-41)	Generates `make` rules, compiles, and prints to a file
`-MM` (on page 1-41)	Generates make rules and compiles
`-Mo` *`filename`* (on page 1-42)	Writes dependency information to *`filename`*. This switch is used in conjunction with the `-ED` or `-MD` options.
`-Mt` *`filename`* (on page 1-42)	Makes dependencies, where the target is renamed as *`filename`*
`-MQ` (on page 1-42)	Generates make rules only; does not compile. No notification when input files are missing
`-map` *`filename`* (on page 1-42)	Directs the linker to generate a memory map of all symbols
`-mem` (on page 1-42)	Enables memory initialization
`-multiline` (on page 1-43)	Enables string literals over multiple lines (default)
`-never-inline` (on page 1-43)	Ignores `inline` keyword on function definitions
`-no-align-branch-lines` (on page 1-43)	Do not align predicted branches to a quad word boundary
`-no-alttok` (on page 1-43)	Does not allow alternative keywords and sequences in sources

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-no-annotate` (on page 1-43)	Disables the annotation of assembly files
`-no-annotate-loop-instr` (on page 1-44)	Disables the production of additional loop annotation information by the compiler (default mode)
`-no-auto-attrs` (on page 1-35)	Directs the compiler not to emit automatic attributes based on the files it compiles.
`-no-bss` (on page 1-44)	Causes the compiler to group global zero-initialized data into the same section as global data with non-zero initializers. Set by default.
`-no-builtin` (on page 1-45)	For certain language extensions, uses generic implementations in preference to intrinsic functions. See "Math Intrinsics" on page 1-143.
`-no-circbuf` (on page 1-45)	Disables the automatic generation of circular buffer code by the compiler
`-no-const-strings` (on page 1-45)	Indicates that string literals should not be qualified as `const`
`-no-defs` (on page 1-45)	Does not define any default preprocessor macros, include directories, library directories, libraries, run-time headers, or keyword extensions
`-no-extra-keywords` (on page 1-46)	Does not define language extension keywords that could be valid C or C++ identifiers
`-no-fp-associative` (on page 1-46)	Does not treat floating-point multiply and addition as an associative
`-no-mem` (on page 1-47)	Disables memory initialization
`-no-multiline` (on page 1-47)	Disables multiple line string literal support
`-no-saturation` (on page 1-47)	Causes the compiler not to introduce saturation semantics when optimizing expressions
`-no-std-ass` (on page 1-47)	Disables any predefined assertions and system-specific macro definitions

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-no-std-def` (on page 1-47)	Disables normal macro definitions; also disables Analog Devices keyword extensions that do not have leading underscores (__)
`-no-std-inc` (on page 1-48)	Searches for preprocessor header files only in the current directory and in directories specified with the `-I` switch
`-no-std-lib` (on page 1-48)	Searches for only those linker libraries specified with the `-l` switch when linking
`-no-threads` (on page 1-48)	Specifies that compiled code does not need to be thread-safe
`-O` (on page 1-48)	Enables code optimizations
`-O [0|1]` (on page 1-49)	Enables/disables code optimizations
`-Oa` (on page 1-49)	Enables automatic function inlining
`-Og` (on page 1-49)	Enables a compiler mode that performs optimizations while still preserving the debugging information
`-Os` (on page 1-50)	Optimizes for code size
`-Ov` *num* (on page 1-50)	Controls speed vs. size optimizations
`-o` *filename* (on page 1-52)	Specifies the output file name
`-P` (on page 1-52)	Preprocesses, but does not compile, the source file; output does not contain `#line` directives
`-PP` (on page 1-53)	Preprocesses and compiles the source file; output does not contain `#line` directives.
`-path-{asm|compiler|lib|link|mem}` *directory* (on page 1-53)	Uses the specified directory as the location of the specified compilation tool (assembler, compiler, librarian, linker, or and memory initializer, respectively)

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-path-install` *directory* (on page 1-53)	Uses the specified directory as the location for all compilation tool
`-path-output` *directory* (on page 1-53)	Specifies the location of non-temporary files
`-path-temp` *directory* (on page 1-53)	Specifies the location of temporary files
`-pch` (on page 1-54)	Generates and uses precompiled header files (`*.pch`)
`-pchdir` *directory* (on page 1-54)	Specifies the location of PCHRepository
`-pedantic` (on page 1-54)	Issues compiler warnings for any constructs that are not ISO/ANSI standard C/C++-compliant
`-pedantic-errors` (on page 1-54)	Issues compiler errors for any constructs that are not ISO/ANSI standard C/C++-compliant
`-pguide` (on page 1-56)	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization
`-pplist` *filename* (on page 1-56)	Outputs a raw preprocessed listing to the specified file
`-proc` *processor* (on page 1-57)	Specifies that the compiler should produce code suitable for the specified processor
`-progress-rep-func` (on page 1-57)	Issues a diagnostic message each time the compiler starts compiling a new function. Equivalent to `-Wwarn=cc1472`.
`-progress-rep-gen-opt` (on page 1-58)	Issues a diagnostic message each time the compiler starts a new generic optimization pass on the current function. Equivalent to `-Wwarn=cc1473`.
`-progress-rep-mc-opt` (on page 1-58)	Issues a diagnostic message each time the compiler starts a new machine-specific optimization pass on the current function. Equivalent to `-Wwarn=cc1473`.
`-R` *directory* (on page 1-58)	Appends directory to the standard search path for source files

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-R-` (on page 1-59)	Removes all directories from the standard search path for source files
`-S` (on page 1-59)	Stops compilation before running the assembler
`-s` (on page 1-59)	Removes debugging information from the output executable file when linking
`-save-temps` (on page 1-59)	Saves intermediate compiler temporary files
`-section` *`id=section_name`* (on page 1-60)	Orders the compiler to place data/program of type "`id`" into the section "`section_name`"
`-show` (on page 1-61)	Displays the driver command-line information
`-si-revision` *`version`* (on page 1-61)	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision.
`-signed-bitfield` (on page 1-62)	Makes the default type for plain `int` bitfields signed
`-signed-char` (on page 1-62)	Makes the default type for `char` signed
`-structs-do-not-overlap` (on page 1-62)	Specifies that struct copies may use "`memcpy`" semantics, rather than the usual "`memmove`" behavior
`-syntax-only` (on page 1-63)	Checks the source code for compiler syntax errors, but does not write any output
`-sysdefs` (on page 1-63)	Defines the system definition macros
`-T` *`filename`* (on page 1-64)	Uses the specified the Linker Description File as control input for linking
`-threads` (on page 1-64)	Specifies that the build and link should be thread-safe
`-time` (on page 1-65)	Displays the elapsed time as part of the output information on each part of the compilation process

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-Umacro` (on page 1-65)	Undefines macro(s)
`-unsigned-bitfield` (on page 1-65)	Makes the default type for plain `int` bitfields unsigned
`-unsigned-char` (on page 1-66)	Makes the default type for `char` unsigned
`-v` (on page 1-66)	Displays version and command-line information for all compilation tools
`-verbose` (on page 1-66)	Displays command-line information for all compilation tools as they process each file
`-version` (on page 1-66)	Displays version information for all compilation tools as they process each file
`-W{error|remark|suppress|warn}` *number* (on page 1-67)	Overrides the default severity of the specified diagnostic-messages (errors, remarks, or warnings)
`-Werror-limit` *number* (on page 1-67)	Stops compiling after reaching the specified number of errors
`-Wremarks` (on page 1-67)	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings
`-Wterse` (on page 1-68)	Issues only the briefest form of compiler warnings, errors, and remarks
`-w` (on page 1-68)	Disables all warnings
`-warn-protos` (on page 1-68)	Issues warnings about functions without prototypes
`-workaround <workaround>` (on page 1-68)	Enables code generator workaround for specific hardware defects
`-write-files` (on page 1-69)	Enables compiler I/O redirection

Table 1-5. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-write-opts` (on page 1-69)	Passes the user options (but not input filenames) via a temporary file
`-xref` *`filename`* (on page 1-69)	Outputs cross-reference information to the specified file.

Table 1-6. C++ Mode Compiler Switches

Switch Name	Description
`-anach` (on page 1-71)	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use
`-check-init-order` (on page 1-72)	Adds run-time checking to the generated code highlighting potential uninitialized external objects.
`-eh` (on page 1-73)	Enables exception handling
`-ignore-std` (on page 1-75)	Disables namespace `std` within the C++ Standard header files.
`-no-anach` (on page 1-75)	Disallows the use of anachronisms that are prohibited by the C++ standard
`-no-demangle` (on page 1-75)	Prevents filtering of any linker errors through the demangler
`-no-eh` (on page 1-75)	Disables exception-handling
`-no-implicit-inclusion` (on page 1-75)	Prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated
`-no-rtti` (on page 1-75)	Disables run-time type information
`-rtti` (on page 1-76)	Enables run-time type information

C/C++ Mode Selection Switch Descriptions

The following command-line switches provide C/C++ mode selection.

-c89

The `-c89` switch directs the compiler to support programs that conform to the ISO/IEC 9899:1990 standard. For greater conformance to the standard, the following switches should be used: `-alttok`, `-const-read-write`, `no-extra-keywords`, and `-pedantic`. (See Table 1-5 on page 1-12.)

-c++

The `-c++` (C++ mode) switch directs the compiler to compile the source file(s) written in ANSI/ISO standard C++ with Analog Devices language extensions. When using this switch, source files with an extension of .c are compiled and linked in C++ mode.

All the standard features of C++ are accepted in the default mode except exception handling and run-time type identification because these impose a run-time overhead that is not desirable for all embedded programs. Support for these features can be enabled with the `-eh` and `-rtti` switches. (See Table 1-6 on page 1-21.)

Exceptions also require modifications to the Linker Description Files; these modifications link against versions of the C++ run-time library that have been built with exceptions support enabled, and link the exception-handling library. There are several new data sections that must be mapped into the DXE as well: `.frt`, `.edt`, `.cht`, and `.gdt`. See the default LDFs included with the release for example modifications.

C/C++ Compiler Common Switch Descriptions

The following command-line switches apply in C and C++ modes.

sourcefile

The *sourcefile* parameter (or parameters) switch specifies the name of the file (or files) to be preprocessed, compiled, assembled, and/or linked. A file name can include the drive, directory, file name, and file extension. The ccts compiler uses the file extension to determine the operations to perform. Table 1-3 on page 1-9 lists the permitted extensions and matching compiler operations.

-@ filename

The -@ filename (command file) switch directs the compiler to read command-line input from *filename*. The specified filename must contain driver options but may also contain source *filenames* and environment variables. It can be used to store frequently used options as well as to read from a file list.

-A name [(<tokens>)]

The -A (assert) switch directs the compiler to assert name as a predicate with the specified tokens. This has the same effect as the #assert preprocessor directive. The following assertions are predefined:

Table 1-7. Predefined Assertions

Assertion	Value
system	embedded
machine	adspts

Table 1-7. Predefined Assertions (Cont'd)

Assertion	Value
cpu	`adspts101` for ADSP-TS101 processor `adspts201` for ADSP-TS201 processor `adspts202` for ADSP-TS202 processor `adspts203` for ADSP-TS203 processor
compiler	`ccts`

The `-A name(value)` switch is equivalent to including

```
#assert name(value)
```

in your source file, and both may be tested in a preprocessor condition in the following manner:

```
#if #name(value)
    // do something
#else
    // do something else
#endif
```

For example, the default assertions may be tested as:

```
#if #machine(adspts)
    // do something
#endif
```

The parentheses in the assertion need quotes when using the `-A` switch, to prevent misinterpretation. No quotes are needed for a `#assert` directive in a source file.

-add-debug-libpaths

The `-add-debug-libpaths` switch prepends the Debug subdirectory to the search paths passed to the linker. The Debug subdirectory, found in each of the silicon-revision-specific library directories, contains variants of certain libraries (for example, system services), which provide additional diagnostic output to assist in debugging problems arising from their use.

Invoke this switch with the **Use Debug System Libraries** radio button located in the VisualDSP++ **Project Options** dialog box, **Link** page, **Processor** category.

-align-branch-lines

The `-align-branch-lines` switch instructs the assembler to align all instruction lines containing a predicted branch to quad-word boundaries. This is the default.

-allow-macs-to-extend-saturation

By default, the compiler will not try to generate multiply-accumulate instructions if the add or subtract is saturating. This switch will override that behaviour. Note that using this switch may result in different output as the multiply-accumulate instruction will saturate to 40 bits for intermediate results and then saturate to 32 bits on extraction from the accumulator.

-alttok

The `-alttok` (alternative tokens) switch directs the compiler to allow alternative operator keywords and digraph sequences in source files. This is the default mode. The switch "-no-alttok" on page 1-43 can be used to disallow such support.

The ANSI C trigraphs sequences are always expanded (even with the `-no-alttok` option), and only digraph sequences are expanded in C source files.

The following operator keywords are enabled by default.

Table 1-8. Keyword Equivalents

Keyword	Equivalent
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

To use alternative tokens in C, use `#include <iso646.h>`.

-always-inline

The `-always-inline` switch instructs the compiler to always attempt to inline any call to a function that is defined with the `inline` qualifier. It is equivalent to applying `#pragma always_inline` to all functions in the module that have the `inline` qualifier See also "-never-inline" on page 1-43.

Invoke this switch with the **Always** radio button located in the **Inlining** area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-annotate-loop-instr

The `-annotate-loop-instr` switch directs the compiler to provide additional annotation information for the prolog, kernel and epilog of a loop. See "Assembly Optimizer Annotations" on page 2-72 for more details on this feature.

-auto-attrs

The `-auto-attrs` (automatic attributes) switch directs the compiler to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See "File Attributes" on page 1-293 for more information about attributes, and what automatic attributes the compiler emits. See also the `-no-auto-attrs` switch (on page 1-44) and the `-file-attr` switch (on page 1-34).

-bss

The `-bss` switch directs the compiler to place global zero-initialized data into a BSS-style section (called "bsz"), rather than into normal global data section. See also "-no-bss" on page 1-44.

-build-lib

The `-build-lib` (build library) switch directs the compiler to use `elfar` (librarian) to produce a library file (`.dlb`) as the output instead of using the linker to produce an executable file (`.dxe`). The `-o` option must be used to specify the name of the resulting library.

-C

The `-C` (comments) switch, which is only active in combination with the `-E` or `-P` switches, directs the C preprocessor to retain comments in its output file.

-c

The `-c` (compile only) switch directs the compiler to compile and/or assemble the source files, but stop before linking. The output is an object file (`.doj`) for each source file.

-char-size-any

The `-char-size-any` switch indicates that the resulting object file should be marked in such a way that it can link against objects marked with any `char` size (8 bit or 32 bit). When linking a project with modules compiled with 8-bit and 32-bit compilers, use the `-char-size-any` switch to avoid any linking error messages.

When a project is linked with a mixture of modules compiled in both word and byte-addressed mode, only one version of the libraries (either word or byte-addressed) can be used. A combination of both is not allowed.

Invoke this switch with the **Allow mixing of sizes** radio buttons located in the VisualDSP++ **Project Options** dialog box, **Compile** category, **Processor** (1) subcategory.

-char-size-{8|32}

The `-char-size-{8|32}` switch specifies that `chars` are 8-bit data items (byte addressing mode) or 32-bit data items (which is the default word addressing mode). Selecting byte addressing mode also sets the macro `__TS_BYTE_ADDRESS` to a value of 1. Use the `-char-size-any` switch to avoid any linking error messages if both 8-bit and 32-bit modules are used in building a project.

Invoke this switch with the **Char size** radio buttons located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Compile** category, **Processor** (1) subcategory.

-const-read-write

The `-const-read-write` switch directs the compiler to specify that constants may be accessed as read-write data (as in ANSI C). The compiler's default behavior assumes that data referenced through `const` pointers never changes.

The `-const-read-write` switch changes the compiler's behavior to match the ANSI C assumption, which is that other `non-const` pointers may be used to change the data at some point.

Invoke this switch with the **Pointers to const may point to non-const data** check box located in the **Constants** area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-const-strings

The `-const-strings` (`const`-qualify strings) swich directs the compiler to mark string literals as `const`-qualified. This is the default behavior. See also the `-no-const-strings` switch (on page 1-45).

Invoke this switch with the **Literal strings are const** check box located in the **Constants** area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-Dmacro[=definition]

The `-D` (define macro) switch directs the compiler to define a macro. If you do not include the optional definition string, the compiler defines the macro as the string '`1`'. If definition is required to be a character string

constant, it must be surrounded by escaped double quotes. Note that the compiler processes `-D` switches on the command line before any `-U` (undefine macro) switches.

This switch can be invoked with the **Definitions:** dialog field located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-debug-types

The `-debug-types` switch builds a `*.h` file directly and writes a complete set of debugging information for the header file. The `-g` option need not be specified with the `-debug-types` switch because it is implied.

For example,

```
ccts -debug-types anyHeader.h
```

Until the introduction of `-debug-types`, the compiler would not accept a `*.h` file as a valid input file. The implicit -g option writes debugging information for only those `typedefs` that are referenced in the program.

The `-debug-types` option provides complete debugging information for all `typedefs` and `structs`.

-default-branch-{np|p}

The `-default-branch-{np|p}` switch instructs the assembler and linker to set the branch behavior to be predictable or non-predictable. The default is the predicted condition.

-double-size-any

The `-double-size-any` switch directs the compiler to mark the resulting object file in such a way that it can link against objects marked with any double size (32 bits and 64 bits).

Invoke this switch with the **Allow mixing of sizes** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Processor (1)** category.

-double-size-{32 | 64}

The `-double-size-32` (double is 32 bits) and `-double-size-64` (double is 64 bits) switches select the storage format that the compiler uses for type `double`. The `-double-size-32` is the default mode.

The C/C++ type `double` poses a special problem for the compiler. The C and C++ languages default to `double` for floating-point constants and many floating-point calculations. If `double` has the customary size of 64 bits, many programs inadvertently use slow-speed 64-bit floating-point emulated arithmetic, even when variables are declared consistently as `float`. To avoid this problem, `ccts` provides a mode in which `double` is the same size as `float`. This mode is enabled with the `-double-size-32` switch and is the default mode.

Representing a `double` using 32 bits gives good performance and provides enough precision for most DSP applications. This, however, does not fully conform to the C and C++ standards. The standards require that `double` maintains 10 digits of precision, which requires 64 bits of storage. The `-double-size-64` switch sets the size of `double` to 64 bits for full standard conformance.

With `-double-size-32`, a `double` is stored in 32-bit IEEE single-precision format and is operated on using fast hardware floating-point instructions. Standard math functions, such as `sin`, also operate on 32-bit values. This

mode is the default and is recommended for most programs. Calculations that need higher precision can be done with the `long double` type, which is always 64 bits.

With `-double-size-64`, a `double` is stored in 64-bit IEEE double precision format and is operated on using slow floating-point emulation software. Standard math functions, such as `sin`, also operate on 64-bit values and are similarly slow. This mode is recommended only for porting code that requires that double have more than 32 bits of precision.

The `-double-size-32` switch defines the `__DOUBLES_ARE_FLOATS__` preprocessor macro, while the `-double-size-64` switch undefines the `__DOUBLES_ARE_FLOATS__` preprocessor macro.

Invoke this switch with the **Double size** radio buttons located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Compile** category, **Processor (1)** subcategory.

-dry

The `-dry` (verbose dry-run) switch directs the compiler to display `ccts` actions, but not to perform them.

-dryrun

The `-dryrun` (terse dry-run) switch directs the compiler to display `ccts` actions, but not to perform them.

-E

The `-E` (stop after preprocessing) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling). The output (preprocessed source code) prints to the standard output stream (`<stdout>`) unless the output file is specified with `-o`. Note that the `-C` switch can only be run in combination with the `-E` switch.

Invoke it with the **Stop after: Preprocessor** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-ED

The `-ED` (run after preprocessing to file) switch directs the compiler to write the output of the C/C++ preprocessor to a file named `original_filename.i`. After preprocessing, compilation proceeds normally.

Invoke this switch with the **Generate preprocessed file** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-EE

The `-EE` (run after preprocessing) switch is similar to the `-E` switch, but it does not halt compilation after preprocessing.

-enum-is-int

The `-enum-is-int` switch ensures that the type of an `enum` is `int`. By default, the compiler defines enumeration types with integral types larger than `int`, if `int` is insufficient to represent all the values in the enumeration. This switch prevents the compiler from selecting a type wider than `int`.

Invoke this switch with the **Enumerated types are always int** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-extra-keywords

The `-extra-keywords` (enable short-form keywords) switch directs the compiler to recognize the Analog Devices keyword extensions to ISO/ANSI standard C and C++. This recognition includes keywords, such as `pm` and `dm`, without leading underscores which could affect conforming ISO/ANSI C and C++ programs. This is the default mode.

The `-no-extra-keywords` switch (on page 1-46) can be used to disallow support for the additional keywords. Table 1-15 on page 1-89 provides a list and a brief description of keyword extensions.

-file-attr name[=value]

The `-file-attr` (file attribute) switch directs the compiler to add the specified attribute name/value pair to all the files it compiles. To add multiple attributes, use the switch multiple times. If "`=value`" is omitted, the default value of "1" will be used. See "File Attributes" on page 1-293 for more information about attributes, and what automatic attributes the compiler emits. See also the `-auto-attrs` switch (on page 1-27) and the `-no-auto-attrs` switch (on page 1-44).

Invoke this switch with the **Additional attributes** text field located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-flags-{asm|compiler|lib|link|mem} switch [,switch2 [,...]]

The `-flags` (command-line input) switch directs the compiler to pass command-line switches to the other build tools, such as:

Table 1-9. Switches Passed to other ADI Tools

Option	Tool
`-flags-asm`	Assembler
`-flags-compiler`	Compiler executable
`-flags-lib`	Library Builder (`elfar.exe`)
`-flags-link`	Linker
`-flags-mem`	Memory Initializer

-force-circbuf

The `-force-circbuf` (circular buffer) switch instructs the compiler to make use of circular buffer facilities, even if the compiler cannot verify that the circular index or pointer is always within the range of the buffer. Without this switch, the compiler's default behavior is conservative, and does not use circular buffers unless it can verify that the circular index or pointer is always within the circular buffer range. See "Circular Buffer Built-In Functions" on page 1-142.

Invoke this switch with the **Even when pointer may be outside buffer range** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-fp-associative

The `-fp-associative` switch directs the compiler to treat floating-point multiplication and addition as an associative. This switch is on by default.

-fp-div-lib

The `-fp-div-lib` switch instructs the compiler to call a run-time library support routine for 32-bit floating-point divides rather that inlining the code. If this switch is specified, a call is planted to a run-time support routine with an entry point named "`__divsf3`" for the divide operation. This routine can be overridden by user-written routines if desired.

Invoke this switch with the **Do not inline float/double divisions** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Processor** category.

-full-version

The `-full-version` (display versions) switch directs the compiler to display version information for build tools used in a compilation.

-g

The `-g` (generate debug information) switch directs the compiler to output symbols and other information used by the debugger.

When the `-g` switch is used in conjunction with the `-O` (enable optimization) switch, the compiler performs standard optimizations. The compiler also outputs symbols and other information to provide limited source-level debugging through VisualDSP++ IDDE. This combination of options provides line debugging and global variable debugging.

When the `-g` and `-O` switches are specified, no debug information is available for local variables and the standard optimizations can sometimes rearrange program code in a way that inaccurate line number information may be produced. For full debugging capabilities, use the `-g` switch without the `-O` switch. See also the `-Og` switch (on page 1-49).

Invoke this switch by selecting the **Generate debug information** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-glite

The `-glite` (lightweight debugging) switch can be used on its own, or in conjunction with any of the `-g`, `-Og` or `-debug-types` compiler switches. When this switch is enabled it instructs the compiler to remove any unnecessary debug information for the code that is compiled.

When used on its own, the switch also enables the `-g` option.

This switch can be used to reduce the size of object and executable files, but will have no effect on the size of the code loaded onto the target.

-H

The `-H` (list headers) switch directs the compiler to output only a list of the files included by the preprocessor via the `#include` directive, without compiling.

-HH

The `-HH` (list headers and compile) switch directs the compiler to output to the standard out a list of the files included by the preprocessor via the `#include` directive. After preprocessing, compilation proceeds normally.

-h[elp]

The `-help` (command-line help) switch directs the compiler to output a list of command-line switches with a brief syntax description.

-I-

The `-I-` (start include directory list) switch establishes the point in the `include` directory list at which the search for header files enclosed in angle brackets begins. Normally, for header files enclosed in double quotes, the compiler searches in the directory containing the current input file; then the compiler reverts back to looking in the directories specified with the `-I` switch and then in the standard include directory.

It is possible to replace the initial search (within the directory containing the current input file) by placing the `-I-` switch at the point on the command line where the search for all types of header file begins. All `include` directories on the command line specified before the `-I-` switch are used only in the search for header files that are enclosed in double quotes.

Note that this switch removes the directory containing the current input file from the `include` directory list.

Invoke this switch with the **Additional include directories** text field located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Preprocessor** category.

-I directory [{,|;} **directory...**]

The `-I` (include search directory) switch directs the C/C++ compiler preprocessor to append the directory (directories) to the search path for `include` files. This option can be specified more than once; all specified directories are added to the search path.

Include files, whose names are not absolute path names and that are enclosed in "..." when included, are searched for in the following directories in this order:

1. The directory containing the current input file (the primary source file or the file containing the `#include`)

2. Any directories specified with the `-I` switch in the order they are listed on the command line
3. Any directories on the standard list:
   `<VDSP++ install dir>/.../include`

If a file is included using the <...> form, this file is only searched for by using directories defined in items 2 and 3 above.

-implicit-pointers

The `-implicit-pointers` (implicit pointer conversion) switch allows a pointer to one type to be converted to a pointer to another without the use of an explicit cast. The compiler produces a discretionary warning rather than an error in such circumstances. This option is not valid when compiling in C++ mode. For example,

```
int *foo(int *a) {
    return a;
}
int main(void) {
    char *p = 0, *r;
    r = foo(p);      /* Bad: will normally give an error. */
    return 0;
}
```

Both the argument to `foo` and the assignment to `r` will be faulted by the compiler. Using `-implicit-pointers` converts these errors into warnings.

Invoke this switch with the **Allow incompatible pointer types** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-include filename

The `-include` (include file) switch directs the preprocessor to process the specified file before processing the regular input file. Any `-D` and `-U` options on the command line are always processed before an `-include` file. Only one `-include` may be given.

-ipa

The `-ipa` (interprocedural analysis) switch directs the compiler to turn on the interprocedural analysis option. This option enables optimization across the entire program, including source files that are compiled separately. Using `-ipa` implicitly enables the `-O` switch. For more information, see "Interprocedural Analysis" on page 1-84.

Invoke this switch by selecting the **Interprocedural Analysis** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-L directory [{,|;} **directory...**]

The `-L` (library search directory) switch directs the linker to append the directory to the search path for library files.

-l library

The `-l` (link library) switch directs the linker to search the library for functions when linking. The library name is the portion of the file name between the `lib` prefix and `.dlb` extension. For example, the `-lc` switch directs the linker to search in the library named "`c`" for functions. This library resides in a file named `libc.dlb`.

All object files should be listed on the command line before listing libraries using the `-l` switch. When a reference to a symbol is made, the symbol definition will be taken from the left-most object or library on the command line that contains the global definition of that symbol. If two

objects on the command line contain definitions of the symbol `x`, `x` will be taken from the left-most object on the command line that contains a global definition of `x`.

If one of the definitions for `x` comes from user objects, and the other from a user library, and the library definition should be overridden by the user object definition, it is important that the user object comes before the library on the command line.

Libraries included in the default `.LDF` file are searched last for symbol definitions.

-M

The `-M` (generate make rules only) switch directs the compiler not to compile the source file, but to output a rule suitable for the make utility, describing the dependencies of the main program file. The format of the make rule output by the preprocessor is:

```
object-file: include-file …
```

-MD

The `-MD` (generate make rules and compile) switch directs the preprocessor to print to a file called `original_filename.d` a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally. See also the `-Mo` switch.

-MM

The `-MM` (generate make rules and compile) switch directs the preprocessor to print to `stdout` a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally.

-Mo filename

The `-Mo filename` (preprocessor output file) switch directs the compiler to use *`filename`* for the output of `-MD` or `-ED` switches.

-Mt name

The `-Mt name` (output make rule for the named source) switch modifies the target of generated dependencies, renaming the target to *`name`*. It only has an effect when used in conjunction with the `-M` or `-MM` switch..

-MQ

The `-MQ` switch directs the compiler not to compile the source file but to output a rule. In addition, the `-MQ` switch does not produce any notification when input files are missing.

-map filename

The `-map` (generate a memory map) switch directs the linker to output a memory map of all symbols. The map file name corresponds to the `filename` argument; if the `filename` argument is `test`, the map file name is `test.xml`. The `.xml` extension is added where necessary.

-mem

The `-mem` (enable memory initialization) switch directs the compiler to invoke the memory initializer after linking the executable file. The MemInit utility can be controlled through the `-flags-mem` switch (on page 1-35).

-multiline

The `-multiline` switch enables a compiler GNU compatibility mode which allows string literals to span multiple lines without the need for a "\" at the end of each line. This is the default mode.

Invoke this switch with the **Allow multi-line character strings** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-never-inline

The `-never-inline` switch instructs the compiler to ignore the `inline` qualifier on function definitions, so that no calls to such functions will be inlined. See also "-always-inline" on page 1-26.

Invoke this switch with the **Never** check box located in the **Inlining** area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-no-align-branch-lines

The `-no-align-branch-lines` (disable alignment of predicted-taken branch lines) switch directs the compiler not to align all instruction lines containing a predicted branch to a quad-word boundary.

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler to not accept alternative operator keywords and digraph sequences in the source files. For more information, see the `-alttok` switch (on page 1-25).

-no-annotate

The `-no-annotate` (disable assembly annotations) switch directs the compiler not to annotate assembly files. The default behavior is that whenever optimizations are enabled, all assembly files generated by the compiler are

annotated with information on the performance of the generated assembly. See "Assembly Optimizer Annotations" on page 2-72 for more details on this feature.

Invoke this switch by clearing the **Generate assembly code annotations** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-no-annotate-loop-instr

The `-no-annotate-loop-instr` switch disables the production of additional loop annotation information by the compiler. This is the default mode.

-no-auto-attrs

The `-no-auto-attrs` (no automatic attributes) switch directs the compiler not to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See "File Attributes" on page 1-293 for more information about attributes, and what automatic attributes the compiler emits. See also the `-auto-attrs` switch (on page 1-27) and the `-file-attr` switch (on page 1-34).

Invoke this switch by clearing the **Auto-generated attributes** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-no-bss

The `-no-bss` switch causes the compiler to keep zero-initialized and non-zero-initialized data in the same data section, rather than separating zero-initialized data into a different, BSS-style section. See also the `-bss` switch (on page 1-27).

-no-builtin

The `-no-builtin` (no built-in functions) switch directs the compiler to use generic implementations in preference to intrinsic functions for certain language extensions. For a list of the extensions that use single machine instructions when `-no-builtin` is not used, see "Math Intrinsics" on page 1-143. This switch also predefines the `__NO_BUILTIN` preprocessor macro.

Invoke this switch by selecting the **Disable builtin functions** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Source Language Settings** category.

-no-circbuf

The `-no-circbuf` (no circular buffer) switch disables the automatic generation of circular buffer code by the compiler. Uses of the `circindex()` and `circptr()` functions (that is, explicit circular buffer operations) are not affected.

Invoke this switch with the **Never** check box located in the **Circular Buffer Generation** area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-no-const-strings

The `-no-const-strings` switch directs the compiler not to make string literals `const` qualified.

-no-defs

The `-no-defs` (disable defaults) switch directs the preprocessor not to define any default preprocessor macros, include directories, library directories, libraries, or run-time headers. It also disables the Analog Devices compiler C/C++ keyword extensions.

-no-extra-keywords

The `-no-extra-keywords` (disable short-form keywords) switch directs the compiler not to recognize the Analog Devices keyword extensions that might affect conformance to ISO/ANSI programs. These extensions include keywords, such as `pm` and `dm`, which may be used as identifiers in conforming programs. The alternate keywords that are prefixed with two leading underscores, such as `__pm` and `__dm`, continue to work. The "-no-extra-keywords" switch (on page 1-46) can be used to explicitly request support for the additional keywords.

Invoke this switch with the **Disable Analog Devices extension keywords** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-no-fp-associative

The `-no-fp-associative` switch directs the compiler NOT to treat floating-point multiplication and addition as an associative.

Invoke this switch with the **Do not treat floating point operations as associative** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-no-fp-minmax

The `-no-fp-minmax` (No floating-point `MIN` or `MAX` instructions) switch prevents the compiler from generating floating-point `MIN` or `MAX` instructions, when it inlines a floating-point comparison. The `MAX` and `MIN` instructions on TigerSHARC will return 0xFFFFFFFF if either input value is a NaN. This can result in behaviour not anticipated or intended by the original user code.

Invoke this switch with the **Do not generate float/double MIN/MAX** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Processor (1)** category.

-no-mem

The `-no-mem` (disable memory initialization) switch directs the compiler not to run the MemInit utility. Note that if you use `-no-mem`, the compiler does not initialize globals and statics.

-no-multiline

The `-no-multiline` switch disables a compiler GNU compatibility mode which allows string literals to span multiple lines without the need for a "\" at the end of each line.

Invoke this switch by clearing the **Allow multi-line character strings** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-no-saturation

The `-no-saturation` switch directs the compiler not to introduce faster operations in cases where the faster operation would saturate (if the expression overflowed) when the original operation would have wrapped the result.

-no-std-ass

The `-no-std-ass` (disable standard assertions) switch prevents the compiler from defining the standard assertions. See the `-A` switch (on page 1-23) for the list of standard assertions.

-no-std-def

The `-no-std-def` (disable standard macro definitions) prevents the compiler from defining any default preprocessor macro definitions.

This switch also disables the Analog Devices keyword extensions which have no leading underscores, such as `pm` and `dm`.

-no-std-inc

The `-no-std-inc` (disable standard include search) switch directs the C preprocessor to limit its search for header files to the current directory and directories specified with `-I` switch.

Invoke this switch by selecting the **Ignore standard include paths** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-no-std-lib

The `-no-std-lib` (disable standard library search) switch directs the linker to limit its search to those libraries specified with the `-l` switch.

-no-threads

The `-no-threads` (disable thread-safe build) switch specifies that all compiled code and libraries used in the build need not be thread-safe. This is the default setting when the `-threads` (enable thread-safe build) switch is not used.

-O

The `-O` (enable optimizations) switch directs the compiler to produce code that is optimized for performance. Optimizations are not enabled by default for the `ccts` compiler.

Invoke this switch by selecting the **Enable optimization** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-O[0|1]

The `-O [0|1]` (enable/disable optimizations) switch allows the compiler to select code optimization mode. Optimizations are not enabled by default for the `ccts` compiler. (Note that the switch settings are numbers—zeros or 1s—while the switch itself is the letter "O.") The switch setting `-O` or `-O1` turns optimization on, while setting `-O0` turns off all optimizations.

Invoke this switch by selecting the **Enable optimization** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-Oa

The `-Oa` (automatic function inlining) switch enables the inline expansion of C/C++ functions, which are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov num` (optimize for speed versus size) switch (on page 1-50). The `-Ov100` setting attempts to optimize purely for speed (and therefore inline a greater number of functions) whereas the `-Ov0` setting attempts to optimize purely for space (and therefore inline fewer functions).

Invoke this switch with the **Automatic** check box located in the **Inlining** area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-Og

The `-Og` switch enables a compiler mode that attempts to perform optimizations while still preserving the debugging information. It is meant as an alternative for those who want a debuggable program but who are also concerned about the performance of their debuggable code.

-Os

The `-Os` (optimize for size) switch directs the compiler to produce code that is optimized for size. This is achieved by performing all optimizations except those that increase code size. The optimizations not performed include loop unrolling, some delay slot filling, and jump avoidance. The compiler also uses a function to save and restore preserved registers for a function instead of generating the more cycle-efficient inline default versions.

-Ov num

For any given optimization, the compiler modifies the code being generated. Some optimizations produce code that will execute in fewer cycles, but which will require more code space. In such cases, there is a trade-off between speed and space.

The `-Ov` *num* (optimize for speed versus size) switch informs the compiler of the relative importance of speed versus size, when considering whether such trade-offs are worthwhile. The *num* variable should be an integer between 0 (purely size) and 100 (purely speed).

The *num* variable indicates a sliding scale between 0 and 100 which is the probability that a linear piece of generated code – a "basic block" – will be optimized for speed or for space. At `-Ov0` all blocks are optimized for space and at `-Ov100` all blocks are optimized for speed. At any point in between, the decision is based upon `num` and how many times the block is expected to be executed – the "execution count" of the block. Figure 1-1 demonstrates this relationship.

For any given optimization where speed and size conflict, the potential benefit is dependent on the execution count: an optimization that increases performance at the expense of code size is considerably more beneficial if applied to the core loop of a critical algorithm than if applied to one-time initialization code or to rarely-used error-handling functions. If code only appears to be executed once, it will be optimized for space. As

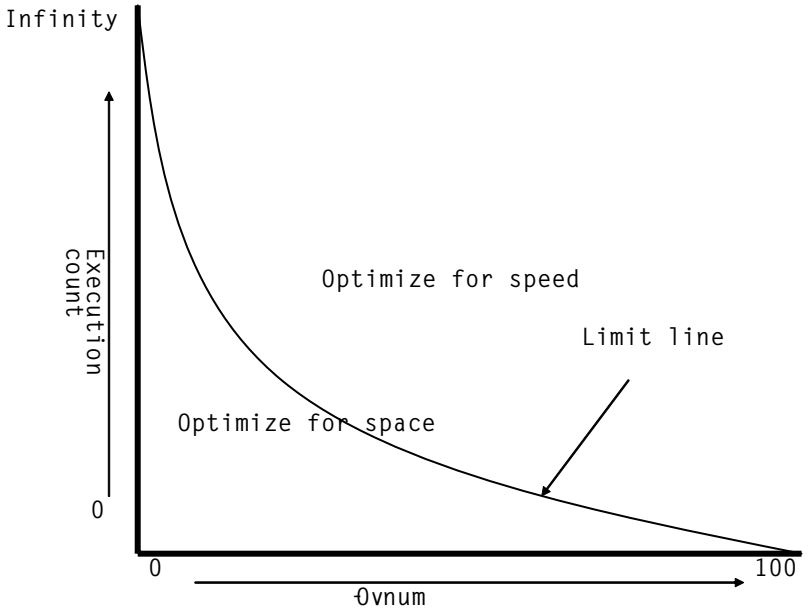

Figure 1-1. -Ov Switch Optimization Curve

its execution count increases, so too does the likelihood that the compiler will consider the code increase worthwhile for the corresponding benefit in performance.

As Figure 1-1 shows, the `-Ov` switch affects the point at which a given execution count is considered sufficient to switch optimization from "for space" to "for speed". Where *num* is a low value, the compiler is biased towards space, so a block's execution count has to be relatively high for the compiler to apply code-increasing transformations. Where *num* has a high value, the compiler is biased towards speed, so the same transformation will be considered valid for a much lower execution count.

The `-Ov` switch is most effective when used in conjuction with profile-guided optimization, where accurate execution counts are available. Without profile-guided optimization, the compiler makes estimates of the relative execution counts using heuristics.

Invoke this switch with the **Optimize for code size/speed** slider located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

For more information, see "Using Profile-Guided Optimization" in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.

-o filename

The `-o` (output file) switch directs the compiler to use *filename* for the name of the final output file.

-overlay

The `-overlay` (program may use overlays) switch will disable the propagation of register information between functions and force the compiler to assume that all functions clobber all scratch registers. Note that this switch will affect all functions in the source file, and may result in a performance degradation. For information on disabling the propagation of register information only for specific functions, see "#pragma overlay" on page 1-202.

-P

The `-P` (omit line numbers) switch directs the compiler to stop after the C preprocessor runs (without compiling) and to omit the `#line` preprocessor directives (with line number information) in the output from the preprocessor. The `-C` switch may be used in combination with the `-P` switch.

-PP

The `-PP` (omit line numbers and compile) switch directs the compiler to omit the `#line` preprocessor directives with line number information from the preprocessor output. After preprocessing, compilation proceeds normally.

-path-{ asm | compiler | lib | link | mem } directory

The `-path-{asm|compiler|lib|link|mem} directory` (tool location) switch directs the compiler to use the specified component in place of the default-installed version of the compilation tool. The component comprises a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, librarian, linker, and memory initializer. Use this switch when overriding the normal version of one or more of the tools. The `-path-{asm|compiler|lib|link|mem}` switch also overrides the directory specified by the `-path-install` switch.

-path-install directory

The `-path-install` (installation location) switch directs the compiler to use the specified directory as the location for all compilation tools instead of the default path. This is useful when working with multiple versions of the tool set.

You can selectively override this switch with the `-path-{asm|compiler|lib|link|mem}` switch.

-path-output directory

The `-path-output` (non-temporary files location) switch directs the compiler to place final output files in the specified directory.

-path-temp directory

The `-path-temp` (temporary files location) switch directs the compiler to place temporary files in the specified directory.

-pch

The `-pch` (precompiled header) switch directs the compiler to automatically generate and use precompiled header files. A precompiled output header has a `.pch` extension attached to the source file name. By default, all precompiled headers are stored in a directory called `PCHRepository`.

Precompiled header files can significantly speed compilation; precompiled headers tend to occupy more disk space.

-pchdir directory

The `-pchdir` (locate PCHRepository) switch specifies the location of an alternative `PCHRepository` for storing and invocation of precompiled header files. If the directory does not exist, the compiler creates it. Note that the `-o` (output) switch does not influence the `-pchdir` option.

-pedantic

The `-pedantic` (ANSI standard warnings) switch causes the compiler to issue warnings for any constructs found in your program that do not strictly conform to the ISO/ANSI standard for C or C++.

The compiler may not detect all noconforming constructs. In particular, the `-pedantic` switch does not cause the compiler to issue errors when Analog Devices keyword extensions are used.

-pedantic-errors

The `-pedantic-errors` (ANSI C errors) switch causes the compiler to issue errors instead of warnings for cases described in the `-pedantic` switch.

-pgo-session session-id

The `-pgo-session` (specify PGO session identifier) switch is used with profile-guided optimization. It has the following effects:

- When used with the `-pguide` switch (on page 1-56), the compiler associates all counters for this module with the session identifier `session-id`.
- When used with a previously-gathered profile (a `.pgo` file), the compiler ignores the profile contents, unless they have the same `session-id` identifier.

This is most useful when the same source file is being built in more than one way (for example, different macro definitions, or for multiple processors) in the same application; each variant of the build can have a different `session-id` associated with it, which means that the compiler will be able to identify which parts of the gathered profile should be used when optimizing for the final build.

If each source file is built only in a single manner within the system (the usual case), then the `-pgo-session` switch is not needed.

Invoke this switch with the **PGO session name** text field located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Profile-Guided Optimization** category.

For more information, see “Using Profile-Guided Optimization” in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.

-pguide

The `-pguide` (PGO) switch causes the compiler to add instrumentation for the gathering of a profile (a `.pgo` file) as the first stage of performing profile-guided optimization.

Invoke this switch with the **Prepare application to create new profile** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Profile-Guided Optimization** category.

For more information, see “Using Profile-Guided Optimization” in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.

-pplist filename

The `-pplist` (preprocessor listing) switch directs the preprocessor to output a listing to the named file. When more than one source file is preprocessed, the listing file contains information about the last file processed. The generated file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler.

Each listing line begins with a key character that identifies its type as:

Table 1-10. Key Characters

Character	Meaning
`N`	Normal line of source
`X`	Expanded line of source
`S`	Line of source skipped by `#if` or `#ifdef`
`L`	Change in source position
`R`	Diagnostic message (remark)
`W`	Diagnostic message (warning)

Table 1-10. Key Characters (Cont'd)

Character	Meaning
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-proc processor

The `-proc processor` (target processor) switch specifies the compiler produces code suitable for the specified processor. Refer to "Supported Processors" for the list of supported TigerSHARC processors.

For example,

```
ccts -proc ADSP-TS201 -o bin\p1.doj p1.asm
```

If no target is specified with the `-proc` switch, the system uses the `ADSP-TS101` setting as a default.

When compiling with the `-proc` switch, the appropriate processor macro is defined as 1.

See also "-si-revision version" on page 1-61 for more information on silicon revision of the specified processor.

-progress-rep-func

The `-progress-rep-func` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing very large source files. It issues a "warning" message each time the compiler starts compiling a new function. The "warning" message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1472`.

-progress-rep-gen-opt

The `-progress-rep-gen-opt` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing a very large, complex function. It issues a "warning" message each time the compiler starts a new generic optimization pass on the current function. The "warning" message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1473`.

-progress-rep-mc-opt

The `-progress-rep-mc-opt` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing a very large, complex function. It issues a "warning" message each time the compiler starts a new machine-specific optimization pass on the current function. The "warning" message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1474`.

-R directory [{:|,}***directory*** **…**]

The `-R` (add source directory) switch directs the compiler to add the specified directory to the list of directories searched for source files.

On Windows platforms, multiple source directories are given as a colon, comma, or semicolon separated list. The compiler searches for the source files in the order specified on the command line. The compiler searches the specified directories before reverting to the current project directory. This option is position-dependent on the command line; that is, it affects only source files that follow the option.

Source files whose file names begin with `/`, `./`, or `../` (or Windows equivalent) and contain drive specifiers (on Windows platforms) are not affected by this option.

-R-

The `-R-` (disable source path) switch removes all directories from the standard search path for source files, effectively disabling this feature.

This option is position-dependent on the command line; it only affects files following it.

-S

The `-S` (stop after compilation) switch directs the compiler to stop compilation before running the assembler. The compiler outputs an assembler file with an `.s` extension.

-s

The `-s` (strip debugging information) switch directs the compiler to remove debugging information (symbol table and other items) from the output executable file during linking.

-save-temps

The `-save-temps` (save intermediate files) switch directs the compiler not to discard intermediate files. The compiler places the intermediate output (`*.i`, `*.is`, `*.s`) files in the current temp directory. See Table 1-3 on page 1-9 for a list of intermediate files.

The location of the saved file is affected by the `-path-output` switch, if provided. That switch sets the path for all "permanent" outputs that do not otherwise have a path set, the object file included.

Invoke this switch with the **Save temporary files** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-section id=section_name[,id=section_name...]

The `-section` switch controls the placement of types of data produced by the compiler. The data is placed into the section "`section_name`" as provided on the command line.

The compiler currently supports the following section identifier:

`code` – controls placement of machine instructions.
Default is `program`.

`data` – controls placement of initialized variable data.
Default is `data1`.

`bsz` – controls placement of zero-initialized variable data.
Default is `bsz`.

`sti` – controls placement of the static C++ class constructor functions.
Default is `program`.

`switch` – controls placement of jump-tables used to implement C/C++ switch statements.
Default is `data1`.

`vtbl` – controls placement of the C++ virtual lookup tables.
Default is `vtbl`.

`vtable` – synonym for `vtbl`

Make sure that the section selected via the command line exists within the `.LDF` file. (Refer to the Linker chapter in the *VisualDSP++ 4.5 Linker and Utilities Manual*.)

For more information, see "Placement of Compiler-Generated Code and Data" on page 1-120.

-show

The `-show` (display command line) switch shows the command-line arguments passed to `ccts`, including expanded option files and environment variables. This option allows you to ensure that command-line options have been passed successfully.

-si-revision version

The `-si-revision` *`version`* (silicon revision) switch directs the compiler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The parameter "*`version`*" represents a silicon revision of the processor specified by the `-proc` switch (on page 1-57).

For example, `ccts -proc ADSP-TS201 -si-revision 1.0 proc.c`

If silicon version "`none`" is used, then no errata workarounds are enabled, whereas specifying silicon version "`any`" will enable all errata workarounds for the target processor.

If the `-si-revision` switch is not used, the compiler will build for the latest known silicon revision for the target processor and any errata workarounds which are appropriate for the latest silicon revision will be enabled.

Run-time libraries built without any errata workarounds are located in the platform's `lib` sub-directory; for example, `TS\lib`. Within the `lib` sub-directory, there are library directories for each silicon revision; these libraries have been built with errata workarounds appropriate for the silicon revision enabled. Note that an individual set of libraries may cover more than one specific silicon revision, so if several silicon revisions are affected by the same errata, then one common set of libraries might be used.

The __SILICON_REVISION__ macro is set by the compiler to two hexadecimal digits representing the major and minor numbers in the silicon revision. For example, 1.0 becomes 0x100 and 10.21 becomes 0xa15.

If the silicon revision is set to "any", the __SILICON_REVISION__ macro is set to 0xffff and if the -si-revision switch is set to "none" the compiler will not set the __SILICON_REVISION__ macro.

The compiler driver will pass the -si-revision <silicon version> switch when invoking another VisualDSP++ tool, for example when the compiler driver invokes the assembler and linker.

Use http://www.analog.com/processors/technicalSupport/hardwareAnomalies.html to get more information on specific anomalies (including anomaly IDs).

-signed-bitfield

The -signed-bitfield (make plain bitfields signed) switch directs the compiler to make bitfields (which have not been declared with an explicit signed or unsigned keyword) signed. This switch does not effect plain one-bit bitfields which are always unsigned. This is the default mode. See also the -unsigned-bitfield switch (on page 1-65).

-signed-char

The -signed-char (make char signed) switch directs the compiler to make the default type for char signed. The compiler also defines the __SIGNED_CHARS__ macro. This is the default mode when the -unsigned-char (make char unsigned) switch is not used.

-structs-do-not-overlap

The -structs-do-not-overlap switch specifies that the source code being compiled contains no structure copies such that the source and the destination memory regions overlap each other in a non-trivial way.

For example, in the statement

```
*p = *q;
```

where `p` and `q` are pointers to some structure type S, the compiler, by default, always ensures that, after the assignment, the structure pointed to by "`p`" contains an image of the structure pointed to by "`q`" prior to the assignment. In the case where `p` and `q` are not identical (in which case the assignment is trivial) but the structures pointed to by the two pointers may overlap each other, doing this means that the compiler must use the functionality of the C library function "`memmove`" rather than "`memcpy`".

It is slower to use "`memmove`" to copy data than it is to use "`memcpy`". Therefore, if your source code does not contain such overlapping structure copies, you can obtain higher performance by using the command-line switch `-structs-do-not-overlap`.

Invoke this switch from the **Structs/classes do not overlap** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Source Language Settings** category.

-syntax-only

The `-syntax-only` (just check syntax) switch directs the compiler to check the source code for syntax errors, but not write any output.

-sysdefs

The `-sysdefs` (system definitions) switch directs the compiler to define several preprocessor macros describing the current user and user's system. The macros are defined as character string constants and are used in functions with null-terminated string arguments.

The following macros are defined if the system returns information for them:

Table 1-11. System Macros

Macro	Description
__HOSTNAME__	The name of the host machine
__MACHINE__	The type of the host machine
__SYSTEM__	The OS name of the host machine
__USERNAME__	The current user's login name
__GROUPNAME__	The current user's group name
__REALNAME__	The current user's real name

The __MACHINE__, __GROUPNAME__, and __REALNAME__ macros are not available on Windows platforms.

-T filename

The -T (Linker Description File) switch directs the linker, when invoked, to use the specified Linker Description File (LDF). If -T is not specified, a default .LDF file is selected based on the processor variant.

-threads

When used, the -threads (enable thread-safe build) switch defines the macro __ADI_THREADS as one (1) at the compile, assemble and link phases of a build. This specifies that certain aspects of the build are to be done in a thread-safe way.

The use of thread-safe libraries is necessary in conjunction with the -threads flag when using the VisualDSP++ Kernel (VDK). The thread-safe libraries can be used with other RTOSs but this requires the definition of various VDK interfaces.

When building applications within VisualDSP++, this switch is added automatically to projects that have VDK support selected.

-time

The `-time` (time the compiler) switch directs the compiler to display the elapsed time as part of the output information on each part of the compilation process.

-Umacro

The `-U` (undefine macro) switch directs the compiler to undefine macros. If you specify a macro name, it is undefined. The compiler processes all `-D` (define macro) switches on the command line before any `-U` (undefine macro) switches.

Invoke this switch by selecting the **Undefines** field in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-unsigned-bitfield

The `-unsigned-bitfield` (make plain bitfields unsigned) switch directs the compiler to make bitfields (which have not been declared with an explicit signed or unsigned keyword) unsigned. This switch does not effect plain one-bit bitfields which are always unsigned. See also the `-signed-bitfield` switch (on page 1-62).

For example, given the declaration

```
struct {
   int a:2;
   int b:1;
   signed int c:2;
   unsigned int d:2;
} x;
```

the `bitfield` values are:

Table 1-12. Bitfield Values

Field	-unsigned-bitfield	-signed-bitfield	Why
x.a	-2..1	0..3	Plain field
x.b	0..1	0..1	One bit
x.c	-2..1	-2..1	Explicit signed
x.d	0..3	0..3	Explicit unsigned

-unsigned-char

The `-unsigned-char` (make char unsigned) switch directs the compiler to make the default type for `char` unsigned. The compiler also undefines the `__SIGNED_CHARS__` preprocessor macro.

-v

The `-v` (version and verbose) switch directs the compiler to display both the version and command-line information for all the compilation tools as they process each file.

-verbose

The `-verbose` (display command line) switch directs the compiler to display command-line information for all the compilation tools as they process each file.

-version

The `-version` (display compiler version) switch directs the compiler to display its version information.

-W {error|remark|suppress|warn} number

The `-W {...}` *`number`* (override error message) switch directs the compiler to override the severity of the specified diagnostic messages (errors, remarks, or warnings). The `num` argument specifies the message to override.

At compilation time, the compiler produces a number for each specific compiler diagnostic message. The `{D}` (discretionary) string after the diagnostic message number indicates that the diagnostic may have its severity overridden. Each diagnostic message is identified by a number that is used across all compiler software releases.

-Werror-limit number

The `-Werror-limit` (maximum compiler errors) switch lets you set a maximum number of errors for the compiler.

-Werror-warnings

The `-Werror-warnings` (treat warnings as errors) switch directs the compiler to treat all warnings as errors, with the result that a warning will cause the compilation to fail.

-Wremarks

The `-Wremarks` (enable diagnostic warnings) switch directs the compiler to issue remarks, which are diagnostic messages that are even milder than warnings.

Invoke this switch by selecting the **Enable remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Warning** selection.

-Wterse

The `-Wterse` (enable terse warnings) switch directs the compiler to issue the briefest form of warnings. This also applies to errors and remarks.

-w

The `-w` (disable all warnings) switch directs the compiler not to issue warnings.

Invoke this switch by selecting the **Disable all warnings and remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Warning** selection.

-warn-protos

The `-warn-protos` (warn if incomplete prototype) switch directs the compiler to issue a warning when it calls a function for which an incomplete function prototype has been supplied. This switch has no effect in C++ mode.

Invoke this switch with the **Function declarations without prototypes** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Warning** category.

-workaround <workaround>[,<workaround>]*

The `-workaround` switch enables code generator workaround for specific hardware defects. Example of a valid workaround: `anomaly-223`.

Use `http://www.analog.com/processors/technicalSupport/hardwareAnomalies.html` to get more information on specific anomalies (including anomaly IDs).

-write-files

The `-write-files` (enable driver I/O redirection) switch directs the compiler driver to redirect the file name portions of its command line through a temporary file. This technique helps with handling long file names, which can make the compiler driver's command line too long for some operating systems.

-write-opts

The `-write-opts` (user options) switch directs the compiler to pass the user options (but not the input *`filenames`*) to the main driver via a temporary file which can help if the resulting main driver command line is too long.

-xref <filename>

The `-xref` (cross-reference list) switch directs the compiler to write cross-reference listing information to the specified file. When more than one source file has been compiled, the listing contains information about the last file processed. For each reference to a symbol in the source program, a line of the form

```
symbol-id name ref-code filename line-number column-number
```

is written to the named file. `symbol-id` represents a unique decimal number for the symbol, and `ref-code` is one of the following characters:

Table 1-13. Possible ref-code Characters

Character	Value
`D`	Definition
`d`	Declaration
`M`	Modification
`A`	Address taken

Table 1-13. Possible ref-code Characters (Cont'd)

Character	Value
U	Used
C	Changed (used and modified)
R	Any other type of reference
E	Error (unknown type of reference

C++ Mode Compiler Switch Descriptions

The following switches apply only to C++.

-anach

The `-anach` (enable C++ anachronisms) directs the compiler to accept some language features that are prohibited by the C++ standard but still in common use. This is the default mode. Use the `-no-anach` switch for greater standard compliance.

The following anachronisms are accepted in the default C++ mode:

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- Assignment to `this` in constructors and destructors is allowed.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as an unnested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a `non-const` type may be initialized from a value of a

different type. A temporary is created; it is initialized from the (converted) initial value, and the reference is set to the temporary.

- A reference to a `non-const` class type may be initialized from an `rvalue` of the `class` type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following statements declare the overload of two functions named `f`.

```
int f(int);
int f(x) char x; { return x; }
```

-check-init-order

It is not guaranteed that global objects requiring constructors are initialized before their first use in a program consisting of separately compiled units. The compiler will output warnings if these objects are external to the compilation unit and are used in dynamic initialization or in constructors of other objects. These warnings are not dependent on the `-check-init-order` switch.

Invoke this switch with the **Check initialization order** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

In order to catch uses of these objects and to allow the opportunity for code to be rewritten, the `-check-init-order` (check initialization order) switch adds run-time checking to the code. This will generate output to `stderr` that indicates uses of such objects are unsafe.

This switch generates extra code to aid development, and should not be used when building production systems.

-eh

The `-eh` (enable exception handling) switch directs the compiler to allow C++ code that contains catch statements and throw expressions and other features associated with ANSI/ISO standard C++ exceptions. When this switch is enabled, the compiler defines the macro `__EXCEPTIONS` to be 1.

This switch also causes the compiler to define `__ADI_LIBEH__` during the linking stage so that appropriate sections can be activated in the `.LDF` file, and the program can be linked with a library built with exceptions enabled.

Object files created with exceptions enabled may be linked with objects created without exceptions. However, exceptions can only be thrown from and caught, and cleanup code executed, in modules compiled with `-eh`.

Invoke this switch with the **C++ exceptions and RTTI** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-full-dependency-inclusion

The `-full-dependency-inclusion` switch ensures that when generating dependency information for implicitly-included `.cpp` files, the `.cpp` file will be re-included. This file is re-included only if the `.cpp` files are

included more than once in the source (via re-inclusion of their corresponding header file). This switch is required only if your C++ sources files are compiled more than once with different macro guards.

Enabling this switch may increase the time required to generate dependencies.

-ignore-std

The `-ignore-std` option is to allow backwards compatibility to earlier versions of VisualDSP C++, which did not use namespace `std` to guard and encode C++ Standard Library names. By default, the header files and Libraries now use namespace `std`.

Invoke this switch by clearing the **Use std:: namespace** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-no-anach

The `-no-anach` (disable C++ anachronisms) switch directs the compiler to disallow some old C++ language features that are prohibited by the C++ standard. See "-anach" on page 1-71 for a full description of these features.

-no-demangle

The `-no-demangle` (disable demangler) switch prevents the compiler from filtering any linker errors through the demangler. The demangler's primary role is to convert the encoded name of a function into a more understandable version of the name.

-no-eh

The `-no-eh` (disable exception handling) directs the compiler to disallow ANSI/ISO C++ exception handling. This is the default mode. See the `-eh` switch (on page 1-73) for more information.

-no-implicit-inclusion

The `-no-implicit-inclusion` switch prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

-no-rtti

The `-no-rtti` (disable run-time type identification) switch directs the compiler to disallow support for `dynamic_cast` and other features of ANSI/ISO C++ run-time type identification. This is the default mode. Use `-rtti` to enable this feature.

-rtti

The `-rtti` (enable run-time type identification) switch directs the compiler to accept programs containing `dynamic_cast` expressions and other features of ANSI/ISO C++ run-time type identification. The switch also causes the compiler to set the macro `__RTTI` to 1. See also the `-no-rtti` switch.

Invoke this switch with the **C++ exceptions and RTTI** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

Data Types and Data Type Sizes

The sizes of intrinsic C/C++ data types are selected by Analog Devices so that normal C/C++ programs execute with hardware-native data types and, therefore, at high speed.

Table 1-14 shows the size used for each of the intrinsic C/C++ data types when in (default) word addressing mode. For details on data type sizes when using byte addressing mode, see "Byte-Addressing Mode" on page 1-91. For information about the `fract` data type, refer to "C++ Fractional Type Support" on page 1-234.

Table 1-14. Default Data Type Sizes for TigerSHARC Processors

Type	Word Size (Bits)	Result of sizeof operator
`char`, `signed char`, `unsigned char`	32	1
`short`, `unsigned short`	32	1
`int`, `unsigned int`	32	1
`long`, `unsigned long`	32	1
`pointer`	32	1
`float`	32	1
`fract`	32 (written with "`r`" suffix) (C++ mode only)	1
`long long int`, `signed long long int`, `unsigned long long int`	64	2
`double`[1]	32 or 64, float	1 or 2
`long double`	64, float	2

1 The double size is 1 when compiled using the `-double-size-32` switch (default), and the `double` size is 2 when compiled using the `-double-size-64` switch. (See on page 1-31.)

When compiling in word addressing mode, the compiler does not support data sizes smaller than a single word location for a processor. For the ADSP-TS101 and ADSP-TS201/202/203 processors, this means that both `short` and `char` have the same size as `int`, unless byte addressing mode is selected. Although 32-bit `chars` are unusual, they do conform to the standard.

Integer Data Types

On any platform, the basic type `int` is the native word size—on TigerSHARC processors, it is 32 bits. Many library functions are available for both 32-bit and 64-bit integers. These functions provide support for the C/C++ data types `int` (or `long int`) and `long long int`. Pointers are the same size as `int` data types. The `long long int` data type provides 64-bit integer.

Floating-Point Data Types

On TigerSHARC processors, the `long` data type is 32 bits, the `float` data type is 32 bits, and `double` is option-selectable for 32-bit or 64-bit data. The C/C++ language tends to default to `double` for constants and for many floating-point calculations. In general, `double` word data types run more slowly than 32-bit data types because they rely largely on software-emulated arithmetic.

Type `double` poses a special problem. Without some special handling, many programs would inadvertently end up using slow-speed, emulated, 64-bit floating-point arithmetic, even when variables are declared consistently as `float`. In order to avoid this problem, Analog provides the "-double-size-{32 | 64}" switch (on page 1-31) that allows you to set the size of `double` to either 32 bits (default) or 64 bits. The 32-bit setting gives good performance and should be acceptable for most DSP programming. However, it does not conform fully to the ANSI C standard.

For a larger floating-point type, `long double` provides 64-bit floating-point arithmetic.

For either size of `double`, the standard `#include` files automatically redefine the math library interfaces so that functions, such as `sin`, can be directly called with the proper size operands.

Access to 64-bit floating-point arithmetic and libraries is always provided via `long double`.

Therefore,

```
float sinf (float);                 /* 32-bit */
double sin (double);                /* 32 or 64-bit */
long double sind (long double);     /* 64-bit */
```

For full descriptions of these functions and their implementation, see Chapter 3, "C/C++ Run-Time Library".

Data Type Alignment

The TigerSHARC run-time model imposes some restrictions on data type alignment that are more stringent than the standard C/C++ alignment restrictions. Refer to "Data Alignment Pragmas" on page 1-181 for more information.

By default, non aggregate data objects are aligned on word boundaries (32 bits) for data types up to and including `long int`. The `long long int` and `long double` (and `double` if `double-size-64` is specified on the command line) data types are aligned on double-word boundaries (64 bits). The extended type `__builtin_quad` is aligned on a four-word boundary (128 bits).

Aggregate types have special rules:

- Top-level arrays are aligned on 4-word boundaries; these are arrays that are not part of any other aggregate type. Statics and automatic arrays are aligned in this way to allow the option of using the TigerSHARC processor's quad-access instructions.

- `struct/union/class` objects are aligned to the maximal alignment of the members and have padding at the end of the structure to maintain alignment in arrays.

Environment Variables Used by the Compiler

The compiler refers to a number of environment variables during its operation, as listed below. The majority of the environment variables identify *path names* to directories. You should be aware that placing network paths into these environment variables may adversely affect the time required to compile applications.

- **PATH**
  This is your System search path, used to locate Windows applications when you run them. Windows uses this environment variable to locate the compiler when you execute it from the command line.

- **TMP**
  This directory is used by the compiler for temporary files, when building applications. For example, if you compile a C file to an object file, the compiler first compiles the C file to an assembly file which can be assembled to create the object file. The compiler usually creates a temporary directory within the TMP directory into which to put such files. However, if the `-save-temps` switch is specified, the compiler creates temporary files in the current directory instead. This directory should exist and be writable. If this directory does not exist, the compiler issues a warning.

- **TEMP**
  This environment variable is also used by the compiler when looking for temporary files, but only if TMP was examined and was not set or the directory that TMP specified did not exist.

- **ADI_DSP**
  The compiler locates other tools in the tool-chain through the VisualDSP++ installation directory, or through the `-path-install` switch. If neither is successful, the compiler looks in `ADI_DSP` for other tools.
- **CCTS_OPTIONS**
  If this environment variable is set, and `CCTS_IGNORE_ENV` is not set, this environment variable is interpreted as a list of additional switches to be appended to the command line. Multiple switches are separated by spaces or new lines. A vertical-bar (|) character may be used to indicate that any switches following it will be processed after all other command-line switches.
- **CCTS_IGNORE_ENV**
  If this environment variable is set, `CCTS_OPTIONS` is ignored.

Optimization Control

The general aim of compiler optimizations is to generate correct code that executes quickly and is small in size. Not all optimizations are suitable for every application or possible all the time. Therefore, the compiler optimizer has a number of configurations, or optimization levels, which can be applied when needed. Each of these levels is enabled by one or more compiler switches (and VisualDSP++ project options) or pragmas.

The following list identifies several optimization levels. The levels are notionally ordered with least optimization listed first and most optimization listed last. The descriptions for each level outline the optimizations performed by the compiler and identify any switches or pragmas required or that have direct influence on the optimization levels performed.

- **Debug**
  The compiler produces debug information to ensure that the object code matches the appropriate source code line. See "-g" on page 1-36 and "-Og" on page 1-49 for more information.

- **Default**
  The compiler does not perform any optimization by default when none of the compiler optimization switches are used (or enabled in VisualDSP++ project options). Default optimization level can be enabled using the `optimize_off` pragma (on page 1-203).

- **Procedural Optimizations**
  The compiler performs advanced, aggressive optimization on each procedure in the file being compiled. The optimizations can be directed to favor optimizations for speed (`-O1` or `-O`) or space (`-Os`) or a factor between speed and space (`-Ov`). If debugging is also requested, the optimization is given priority so the debugging functionality may be limited. See "-O[0|1]" on page 1-49, "-Os" on page 1-50, "-Ov num" on page 1-50 and "-Og" on page 1-49. Procedural optimizations for speed and space (`-O` and `-Os`) can be enabled in C/C++ source using the pragma `optimize_{for_speed|for_space}` (for more information on optimization pragmas, see on page 1-203).

- **Profile-Guided Optimizations (PGO)**
  The compiler performs advanced aggressive optimizations using profiler statistics (`.pgo` files) generated from running the application using representative training data. PGO can be used in conjunction with IPA and Automatic Inlining. See "-pguide" on page 1-56 for more information.

The most common scenario in collecting PGO data is to set up one or more simple *File to Device* streams where the *File* is a standard ASCII stream input file and the *Device* is any stream device supported by the simulator target, such as memory and peripherals. The PGO process can be broken down into the execution of one or more *data sets*, where a data set is the association of zero or more input streams with one and only one `.pgo` output file. The user can create, edit and delete the data sets through the IDDE and then run the data sets with the click of one button to produce an optimized application. The PGO operation is handled via a new PGO submenu added to the top-level **Tools** menu:
Tools -> PGO -> Manage Data Sets.

For more information, see "Using Profile-Guided Optimization" in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.

Note the requirement for allowing command-line arguments in your project when using PGO. For further details refer to "Support for argv/argc" on page 1-264.

- **Automatic Inlining**
  The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code. It does this when it has determined that doing so reduces execution time. How aggressively the compiler performs automatic inlining is controlled using the `-Ov` switch. Automatic inlining is enabled using the `-Oa` switch and additionally enables procedural optimizations (`-O`). See "-Oa" on page 1-49, "-Ov num" on page 1-50, "-O[0|1]" on page 1-49 and "Function Inlining" on page 1-95 for more information.

When remarks are enabled, the compiler produces a remark to indicate each function that is inlined.

- **Interprocedural Optimizations**
  The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. The *interprocedural analysis* (IPA) is enabled using the `-ipa` switch and additionally enables procedural optimizations (`-O`). See "Interprocedural Analysis" on page 1-84, "-ipa" on page 1-40 and "-O[0|1]" on page 1-49 for more information.

The compiler optimizer attempts to vectorize loops when it is safe to do so. When IPA is used it can identify additional safe candidates for vectorization which might not be classified as safe at a Procedural Optimization level. Additionally, there may be other loops that are known to be safe candidates for vectorization which can be identified to the compiler with use of various pragmas. (See "Loop Optimization Pragmas" on page 1-187.)

Using the various compiler optimization levels is an excellent way of improving application performance. However consideration should be given to how applications are written so that compiler optimizations are given the best opportunity to be productive. These issues are the topic of Chapter 2, "Achieving Optimal Performance from C/C++ Source Code".

Interprocedural Analysis

The `ccts` compiler has a capability called *interprocedural analysis* (IPA), a mechanism that allows the compiler to optimize across translation units instead of within just one translation unit. This capability effectively allows the compiler to see all of the source files that are used in a final link at compilation time and make use of that information when optimizing.

Interprocedural analysis is enabled by selecting the **Interprocedural Analysis** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category, or by specifying the `-ipa` command-line switch. (See "-ipa" on page 1-40.)

The `-ipa` switch automatically enables the `-O` switch to turn on optimization.

Use of the `-ipa` switch generates additional files along with the object file produced by the compiler. These files have `.ipa` and `.opa` filename extensions and should not be deleted manually unless the associated object file is also deleted.

All of the `-ipa` optimizations are invoked after the initial link, whereupon a special program called the prelinker reinvokes the compiler to perform the new optimizations.

Because a file may be recompiled by the prelinker, you cannot use the `-S` option to see the final optimized assembler file when `-ipa` is enabled. Instead, use the `-save-temps` switch (on page 1-59), so that the full compile/link cycle can be performed first.

Interaction with Libraries

When IPA is enabled, the compiler examines all of the source files to build up usage information about all of the function and data items. It then uses that information to make additional optimizations across all of the source files. One of these optimizations is to remove functions that are never called. This optimization can significantly reduce the overall size of the final executable.

Because IPA operates only during the final link, the `-ipa` switch has no benefit when initially compiling source files to object format for inclusion in a library. Although IPA generates usage information for potential additional optimizations at the final link stage, as normal, neither the usage information nor the module's source file are available when the linker includes a module from a library. Each library module has been compiled

to the normal `-O` optimization level, but the prelinker cannot access the previously-generated additional usage information for an object in a library. Therefore, IPA cannot exploit the additional information associated with a library module.

If a library module makes references to a function in a user module in the program, this will be detected during the initial linking phase, and IPA will not eliminate the function. IPA will also not make any assumptions about how the function may be called, so the function may not be optimized as effectively as if all references to it were in source code visible to IPA.

C/C++ Compiler Language Extensions

The compiler supports a set of extensions to the ANSI standard for the C and C++ languages. These extensions add support for DSP hardware and allow some C++ programming features when compiling in C mode. The extensions are also available when compiling in C++ mode.

This section contains:

- "Byte-Addressing Mode" on page 1-91
- "Function Inlining" on page 1-95
- "Inline Assembly Language Support Keyword (asm)" on page 1-100
- "64-Bit Integer Support (long long)" on page 1-114
- "Quad-Word Support" on page 1-115
- "Memory Support Keywords (pm dm)" on page 1-115
- "__regclass Construct" on page 1-117
- "Bank Type Qualifiers" on page 1-118
- "Placement Support Keyword (section)" on page 1-119
- "Placement of Compiler-Generated Code and Data" on page 1-120
- "Boolean Type Support Keywords" on page 1-121
- "Pointer Class Support Keyword (restrict)" on page 1-121
- "Variable-Length Array Support" on page 1-122
- "Non-Constant Aggregate Initializer Support" on page 1-124
- "Indexed Initializer Support" on page 1-124
- "Compiler Built-In Functions" on page 1-126

- "Pragmas" on page 1-179
- "C++ Style Comments" on page 1-233
- "C++ Fractional Type Support" on page 1-234
- "GCC Compatibility Extensions" on page 1-237
- "Preprocessor-Generated Warnings" on page 1-245

The additional keywords that are part of these C/C++ extensions do not conflict with any ISO/ANSI C/C++ keywords. The formal definitions of these extension keywords are prefixed with a leading double underscore (__). Unless the `-no-extra-keywords` command-line switch is used, the compiler defines the shorter forms of the keyword extension that omits the leading underscores. See "-extra-keywords" on page 1-34 for more information.

This section describes only the shorter forms of the keyword extensions, but in most cases you can use either form in your code. For example, all references to the `inline` keyword in this text appear without the leading double underscores, but you can use `inline` or `__inline` interchangeably in your code.

You might need to use the longer forms (such as `__inline`) exclusively if porting a program that uses the extra Analog Devices keywords as identifiers. For example, a program might declare local variables, such as `pm` or `dm`. In this case, you should use the `-no-extra-keywords` switch, and if you need to declare a function as inline, or allocate variables to memory spaces, you can use `__inline` or `__pm`/`__dm`, respectively.

Table 1-15 provides a list and a brief description of keyword extensions. Table 1-16 provides a list and a brief description of operational extensions. Both tables direct you to sections of this chapter that document each extension in more detail.

Table 1-15. Keyword Extensions

Keyword extensions	Description
`Byte Addressing`	Use the "-char-size-{8|32}" switch to select the byte addressing mode. For more information, see "Byte-Addressing Mode" on page 1-91.
`inline (function)`	Directs the compiler to integrate the function code into the code of the callers.For more information, see "Function Inlining" on page 1-95.
`asm()`	Directs the compiler to code TigerSHARC assembly language instructions within a C or C++ function. For more information, see "Inline Assembly Language Support Keyword (asm)" on page 1-100.
`long long [int]`	Provides 64-bit integer support, both signed and unsigned. For more information, see "64-Bit Integer Support (long long)" on page 1-114.
`dm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to Data Memory. For more information, see "Memory Support Keywords (pm dm)" on page 1-115.
`pm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to Program Memory. For more information, see "Memory Support Keywords (pm dm)" on page 1-115.
`bank("string")`	Specifies the name of the memory introduced by the user. For more information, see "Bank Type Qualifiers" in Chapter 1, Compiler.
`section("string")`	Specifies the section in which an object or function is placed. For more information, see "Placement Support Keyword (section)" on page 1-119.
`__regclass(unit)`	Gives the compiler hints about which TigerSHARC unit to use for a particular calculation. For more information, see "__regclass Construct" on page 1-117.

Table 1-15. Keyword Extensions (Cont'd)

Keyword extensions	Description
`bool, true, false`	A Boolean type. For more information, see "Boolean Type Support Keywords" on page 1-121.
`restrict` keyword	Specifies restricted pointer features. For more information, see "Pointer Class Support Keyword (restrict)" on page 1-121.

Table 1-16. Operational Extensions

Operation extensions	Description
Variable-length arrays	Support for variable length arrays lets you use automatic arrays whose length is not known until runtime. For more information, see "Variable-Length Array Support" on page 1-122.
Non-constant initializers	Support for non-constant initializers lets you use non-constants as elements of aggregate initializers for automatic variables. For more information, see "Non-Constant Aggregate Initializer Support" on page 1-124.
Indexed initializers	Support for indexed initializers lets you specify elements of an aggregate initializer in arbitrary order. For more information, see "Indexed Initializer Support" on page 1-124.
Preprocessor-generated warnings	Support for generating warning messages from the preprocessor. For more information, see "Preprocessor-Generated Warnings" on page 1-245.
C++-style comments	Support for C++-style comments in C programs. For more information, see "C++ Style Comments" on page 1-233.
`fract` data type (C++ mode)	(Provides support for the fractional data type, fractional and saturated arithmetic.For more information, see "C++ Style Comments" on page 1-233.
Quad word support	Provides support for a quad word (128 bit) data type. For more information, see "Quad-Word Support" on page 1-115.

Byte-Addressing Mode

To select the byte-addressing mode of operation for the TigerSHARC compiler, the compiler flag `-char-size-8` (on page 1-28) should be selected both for compilation and link stages. This also has the effect of defining the macro `__TS_BYTE_ADDRESS` with a value of 1, both when compiling and when invoking the linker. The default `.LDF` file, when the `__TS_BYTE_ADDRESS` macro is so defined, links against the byte address versions of the run-time libraries.

When using byte-addressing mode, it is vital that the correct C/C++ header files are included in all source files. (See "Libraries Used in Byte-Addressing Mode" on page 1-94 for more information.)

sizeof() Operator Types and Sizes

Table 1-17 shows the sizes in bits and the value returned by the `sizeof()` operator for the fundamental types that are supported by the compiler in byte-addressing mode.

Table 1-17. Byte-Addressing Types and Bit Lengths

Type	Size(Bits)	sizeof(T)
`char, signed char, unsigned char`	8	1
`short, unsigned short`	16	2
`int, unsigned int`	32	4
`long, signed long, unsigned long`	32	4
`long long, signed long long, unsigned long long`	64	8
`float`	32	4
`double`	32	4
`long double`	64	8

Table 1-17. Byte-Addressing Types and Bit Lengths

Type	Size(Bits)	sizeof(T)
`__builtin_quad`	128	16
`pointers`	32	4

Pointers

The pointer representation uses the low-order 30 bits to address the word and the high-order two bits to address the byte within the word. Due to the pointer implementation, the address range in byte-addressing mode is `0x00000000` to `0x3FFFFFFF`. As noted elsewhere, the run-time support, including memory allocation, must be either the byte-addressing support or the word-addressing support; therefore, there is no 32-bit pointer restriction.

The main advantage of using the high-order bits to address the bytes within the word as opposed to using the low-order bits is that all pointers that address word boundaries are compatible with existing code. This choice means there is no performance loss when accessing 32-bit items.

A minor disadvantage with this representation is that address arithmetic is slower than using low-order bits to address the bytes within a word when the computation might involve part-word offsets.

Alignment of Objects

Within a structure, members of the fundamental types are aligned on a multiple of their size. Structures are aligned on the strictest alignment of any of their members, but are always aligned to at least 32 bits.

As always, the size of a structure is a multiple of its alignment, so this last restriction leads to subtle differences from C implementations on Windows and UNIX® platforms.

For example,

```
struct A { short a; short b; short c; };
sizeof(struct A)      // is 8 not 6 !!!.
```

Entire variables are aligned to at least 32 bits. They are more strictly aligned if their type requires it. Entire array variables are aligned to 128 bits.

Initializations

Initializations that require a part-word address to be computed at compile or link-time is not supported. For example,

```
short a[20];
short *p = a + 3;      /* illegal */
```

The compiler reports these initializations as errors.

Pragmas Used in Byte-Addressing Mode

Support for the alignment pragma has additional considerations in byte addressing mode. The following pragma

```
#pragma align n
```

is supported, although the alignment argument `n` is in bytes rather than in words and is only allowed to specify alignment on a word boundary. For more information on this pragma, refer to "Data Alignment Pragmas" on page 1-181.

Performance Issues

It is quite expensive to have unoptimized loads, stores of part-words and arithmetic involving pointers to part-words. The arithmetic requires rotating the address, performing the addition or subtraction and then rotating the result. A load of a part-word must first load the word, test the

high-order bits of the address, and then extract the correct part-word. A store must load the entire word, deposit the part-word into it, and then write back the whole word.

Consequently, both the performance of the generated code and the size of the generated code depends upon how successful the compiler is at optimizing part-word load and stores and pointer arithmetic.

Wherever possible, use `integer` types instead of `short` or `char` data types to improve the quality of the generated code.

Libraries Used in Byte-Addressing Mode

The run-time support libraries used in byte-addressing mode are independent from those used in word addressing mode. It is always necessary to include the appropriate system header files when compiling in both word-addressing and byte-addressing modes. The reason for this is that some of the underlying entry-point names within the supplied run-time support libraries can differ depending on the compilation mode selected; for example, the I/O support differs for the 64-bit and 32-bit `doubles`. If the header files containing the prototypes for referenced run-time support functions are not included, this could result in unresolved symbols at link time or incorrect run-time behavior. For example, include `<stdio.h>` to use `printf()`, or `<stdlib.h>` to use `malloc()`.

The memory management routines allocate memory in units of `sizeof(char)` although the implementation only allocates and deallocates complete words. Requests are rounded up to the next word boundary where appropriate, so that all actual memory claimed and freed are in multiples of words.

The run-time support libraries for byte-addressing mode are not compatible with the word addressing support libraries. Applications have to exclusively use either the byte-addressing support libraries or the word-addressing support libraries and can not use a combination of both.

Include Files

The header files are modified to represent the correct sizes and bounds for the data types under the `__TS_BYTE_ADDRESS` macro which is defined by the compiler under byte-addressing mode.

Function Inlining

The `inline` keyword directs the compiler to integrate the code for the function you declare as `inline` into the code of its callers. Inline function support and the `inline` keyword is a standard feature of C++; the compiler provides this keyword as a C extension.

This keyword eliminates the function call overhead and increases the speed of your program's execution. Argument values that are constant and that have known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

The following example shows a function definition that uses the `inline` keyword.

```
inline int max3 (int a, int b, int c) {
   return max (a, max(b, c));
}
```

The compiler can decide not to inline a particular function declared with the `inline` keyword, with a diagnostic remark `cc1462` issued if the compiler chooses to do this. The diagnostic can be raised to a warning by use of the `-Wwarn` switch. For more information, see "-W {error|remark|suppress|warn} number" on page 1-67.

Function inlining can also occur by use of the `-Oa` (automatic function inlining) switch (For more information, see "-Oa" on page 1-49.), which enables the inline expansion of C/C++ functions that are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov` (optimize for speed versus size) switch.

The compiler follows a specific order of precedence when determining whether a call can be inlined. The order is:

1. If the definition of the function is not available (for example, a call to an external function), the compiler cannot inline the call.
2. If the `-never-inline` switch has been specified ("-never-inline" on page 1-43), the compiler will not inline the call. If the call is to a function that has `#pragma always_inline` specified ("Inline Control Pragmas" on page 1-204), a warning will also be issued.
3. If the call is to a function that has `#pragma never_inline` specified, the call will not be inlined.
4. If the call is via a pointer-to-function, the call will not be inlined unless the compiler can prove that the pointer will always point to the same function definition.
5. If the call is to a function that has a variable number of arguments, the call will not be inlined.
6. If the module contains `asm` statements at global scope (outside function definitions), the call may not be inlined because the `asm` statement restricts the compiler's ability to reorder the resulting assembly output.
7. If the call is to a function that has `#pragma always_inline` specified, the call is inlined. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.
8. If the call is to a function that has the `inline` qualifier, and the `-always-inline` switch has been specified, the compiler will inline the call. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.

9. If the call is to a function that has the `inline` qualifier and optimization is enabled, the called function will be compared against the current speed/size ratio limits for code size and stack size. The calling function will also be examined against these limits. Depending on the limits and the relative sizes of the caller and callee, the inlining may be rejected.

10. If the call is to a function that does not have the `inline` qualifier, but the `-Oa` switch has been specified to enable automatic inlining, the called function will be considered as a possible candidate for inlining, according to the current speed/size ratio limits, as if the `inline` qualifier were present.

The compiler bases its code-related speed/size comparisons on the `-Ov` switch. When `-Ov` is in the range 1...100, the compiler performs a calculation upon the size of the generated code using the `-Ov` value, and this will determine whether the generated code is "too large" for inlining to occur. When `-Ov` has the value 1, only very small functions are considered small enough to inline; when `-Ov` has the value 100, larger functions are more likely to be considered suitable as well.

When `-Ov` has the value 0, the compiler is optimizing for space. The speed/space calculation will only accept a call for inlining if it appears that the inlining is likely to result in less code than the call itself would. (This is an approximation, since the inlining process is a high-level optimization process, before actual machine instructions have been selected.)

The inlining process also considers the required stack size while inlining. A function that has a local array of 20 integers needs such an array for each inlined invocation, and if inlined many times, the cumulative effect on overall stack requirements can be significant. Consequently, the compiler considers both the stack space required by the called function, and the total stack space required by the caller; either may reach a limit at which the compiler determines that inlining the call would not be beneficial. The stack size analysis is not subject to the `-Ov` switch.

Inlining and Optimization
The inlining process operates regardless of whether optimization has been selected (although if optimization is not enabled, then inlining will only happen when forced by `#pragma always_inline` or the `-always-inline` switch). The speed/size calculation still has an effect, although an optimized function is likely to have a different size from a non-optimized one; which is smaller (and therefore more likely to be inlined) is dependent on the kind of optimization done.

A non-optimized function has loads and stores to temporary values which are optimized away in the optimized version, but an optimized function may have unrolled or vectorized loops with multiple variants, selected at run-time for the most efficient loop kernel, so an optimized function may run faster, but not be smaller.

Given that the optimization emphasis may be changed within a module – or even turned off completely – by the optimization pragmas, it is possible for either, both, or neither of the caller and callee to be optimized. The inlining process still operates, and is only affected by this in as far as the speed/size ratios of the resulting functions are concerned.

Inlining and Out-of-Line Copies
If a function is static (that is, private to the module being compiled) and all calls to that function are inlined, then there are no calls remaining that are not inline. Consequently, the compiler does not generate an out-of-line copy for the function, thus reducing the size of the resulting application.

If the address of the function is taken, it is possible that the function could be called through that derived pointer, so the compiler cannot guarantee that all calls have been accounted for. In such cases, an out-of-line copy will always be generated.

A function declared `inline` must be defined (its body must be included) in every file in which the function is used. This is normally done by placing the `inline` definition in a header file. Usually it is also declared static.

Inlining and Global asm Statements
Inlining imposes a particular ordering on functions. If functions A and B are both marked as inline, and each calls the other, only one of the `inline` qualifiers can be followed. Depending on which the compiler chooses to apply, either A will be generated with inline versions of B, or B will be generated with inline versions of A. Either case may result in no out-of-line copy of the inlined function being generated. The compiler reorders the functions within a module to get the best inlining result. Functionally, the code is the same, but this affects the resulting assembly file.

When global `asm` statements are used with the module, between the function definitions, the compiler cannot do this reordering process, because the `asm` statement might be affecting the behavior of the assembly code that is generated from the following C function definitions. Because of this, global `asm` statements can greatly reduce the compiler's ability to inline a function call.

Inlining and Sections
When inlining, any section directives or pragmas on the function definitions are ignored. For example,

```
section("secA") inline int add(int a, int b) { return a + b; }
section("secB") int times_two(int a) { return add(a, a); }
```

Although `add()` and `times_two()` are to be generated into different code sections, this is ignored during the inlining process, so if the code for `add()` is inlined into `times_two()`, the inlined copy appears in section "`secB`" rather than section "`secA`". Only when out-of-line copies are generated (if necessary) does the compiler make use of any section directive or pragma applied to the out-of-line copy of the inlined function.

Inline Assembly Language Support Keyword (asm)

The compiler `asm()` construct allows you to code TigerSHARC assembly language instructions within a C or C++ function. The `asm()` construct is useful in expressing assembly language statements that cannot be expressed easily or efficiently with C/C++ constructs.

Using `asm()`, you can code complete assembly language instructions and specify the operands of the instruction using C expressions. When specifying operands with a C/C++ expression, you do not need to know which registers or memory locations contain C or C++ variables.

The compiler *does not analyze* code defined with the `asm()` construct— it passes this code directly to the assembler. The compiler *does* perform substitutions for operands of the formats `%0` through `%9`. However, it passes *everything else* through to the assembler without reading or analyzing it.

The `asm()` constructs are executable statements, and as such, may not appear before declarations within C/C++ functions.

The `asm()` constructs may also be used at global scope, outside function declarations. Such `asm()` constructs are used to pass declarations and directives directly to the assembler. They are not executable constructs, and may not have any inputs or outputs, or affect any registers.

When optimizing, the compiler sometimes changes the order in which generated functions appear in the output assembly file. However, if global-scope `asm` constructs are placed between two function definitions, the compiler ensures that the function order is retained in the generated assembly file. Consequently, function inlining may be inhibited.

A simplified `asm()` construct without operands takes the form of

```
asm("nop;;");
```

The complete assembly language instruction, enclosed in quotes, is the argument to `asm()`.

The compiler generates a label before and after inline assembly instructions when generating debug code (the `-g` switch on page 1-36). These labels are used to generate the debug line information used by the debugger. If the inline assembler inserts conditionally assembled code, an undefined symbol error is likely to occur at link time. For example, the following code could cause undefined symbols if `MACRO` is undefined:

```
asm("#ifdef MACRO");
asm(" // assembly statements");
asm("#endif");
```

If the inline assembler changes the current section and thereby causes the compiler labels to be placed in another section, such as a data section (instead of the default code section), then the debug line information is incorrect for these lines.

If the assembly construct contains assembly code that can alter the values of status registers, these status registers must be included in the list of clobbered registers. Information on the status registers, and on code which may affect them, can be found in the relevant programming reference.

Using `asm()` constructs with operands requires some additional syntax. The construct syntax is described in:

- “Assembly Construct Template”
- “Assembly Construct Operand Description” on page 1-105
- “Assembly Constructs With Multiple Instructions” on page 1-110

- "Assembly Construct Reordering and Optimization" on page 1-111
- "Assembly Constructs With Input and Output Operands" on page 1-112
- "Assembly Constructs and Flow Control" on page 1-113
- "Guidelines on the Use of asm() Statements" on page 1-113

Assembly Construct Template

Using `asm()` constructs, you can specify the operands of the assembly instruction that employ C expressions. You do not need to know which registers or memory locations contain C variables.

ASM() Construct Syntax:

Use the following general syntax for your `asm()` constructs.

```
asm(
   template
   [:[constraint(output operand)[,constraint(output operand)…]]
     [:[constraint(input operand)[,constraint(input operand)…]]
             [:clobber]]]
);
```

The syntax elements are defined as:

- **template**
- The template is a string containing the assembly instruction(s) with `%number` indicating where the compiler should substitute the operands. Operands are numbered in order of appearance from left to right, starting at 0. Separate multiple instructions with a semico-

lon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see "Assembly Constructs With Multiple Instructions" on page 1-110.

- **constraint**
- The constraint is a string that directs the compiler to use certain groups of registers for the input and output operands. Enclose the constraint string within double quotes. For more information on operand constraints, see "Assembly Construct Operand Description" on page 1-105.
- **output operand**
  The output operands are the names of a C/C++ variables that receives output from a corresponding operand in the assembly instructions.
- **input operand**
  The input operand is a C or C++ expression that provides an input to a corresponding operand in the assembly instruction.
- **clobber**
  The clobber notifies the compiler that a list of registers is overwritten by the assembly instructions. Use lowercase characters to name clobbered registers. Enclose each register name within double quotes, and separate the quoted register names with commas. It is vital that any register overwritten by an assembly instruction is included in the clobber list.

ASM() Construct Syntax Rules

These rules apply to assembly construct template syntax:

- The template is the only mandatory argument to the `asm()` construct. All other arguments are optional.

- An operand constraint string followed by a C expression in parentheses describes each operand. For output operands, it must be possible to assign to the expression—that is, the expression must be legal on the left side of an assignment statement.
- A colon separates the template from the first output operand, the last output operand from the first input operand, and the last input operand from the clobbered registers. If there are no output operands and there are input operands, there must be two consecutive colons separating the assembly template from the input operands. Add a space between adjacent colon field delimiters in order to avoid a clash with the C++ "::" reserved global resolution operator.
- A comma separates operands and registers within arguments.
- The number of operands in arguments must match the number of operands in your template.
- The maximum permissible number of operands is ten (`%0`, `%1`, `%2`, `%3`, `%4`, `%5`, `%6`, `%7`, `%8`, and `%9`).

The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. The compiler does not parse the assembler instruction template, does not interpret the template, and does not verify whether the template contains valid input for the assembler.

ASM() Construct Template Example

The following example shows how to apply the `asm()` construct template to the TigerSHARC assembly language `abs` instruction:

```
{
int result, x;
...
asm (
    "%0 = %1;;":
    "=&k" (result):
```

```
        "k" (x));
}
```

In the previous example, note the following points:

- The template is "`%0 = %1;;`". The `%0` is replaced with operand zero `(result)`, the first operand. The `%1` is replaced with operand one `(avar)`.
- The output operand is the C/C++ variable `result`. The letter `k` is the *operand constraint* for the variable. This constrains the output to an `IALU` K-register. The compiler generates code to copy the output from the `IALU` register to the variable `result`, if necessary. The "=" in `=&k` indicates that the operand is an output. The & constraint ensures that the register assigned to the output cannot be the same register as the register assigned to the input.
- The input operand is the C/C++ variable `(x)`. The letter `k` is the operand constraint position for this variable constrains `x` to an IALU K-register. If `x` is is stored in different kinds of registers or in memory, the compiler generates code to copy the values into an K-register before the `asm()` construct uses them.

Assembly Construct Operand Description

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. The compiler has to obtain several pieces of information in order to know how to assign registers to operands. This information is conveyed with an operand constraint. The compiler needs to know what kind of registers the assembly instructions can operate on, so it can allocate the correct register type.

You convey this information with a letter in the operand constraint string which describes the class of allowable registers.

Table 1-18 on page 1-109 describes the correspondence between constraint letters and register classes. Table 1-19 on page 1-109 describes constraint operators.

The use of any letter not listed in Table 1-18 results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

To assign registers to the operands, the compiler must also be told which operands in an assembly language instruction are inputs, which are outputs, and which outputs may not overlap inputs. The compiler is told this in three ways.

- The output operand list appears as the first argument after the assembly language template. The list is separated from the assembly language template with a colon. The input operands are separated from the output operands with a colon and always follow the output operands.
- The operand constraints describe which registers are modified by an assembly language instruction. The = in `=constraint` indicates that the operand is an output; all output operand constraints must use =.
- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output operand has the `=&` constraint modifier. This situation can occur because the compiler assumes that the inputs are consumed before the outputs are produced.

This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `=&` for each output operand that must not overlap an input or supply an "`&`" for the input operand.

Operand constraints indicate what kind of operand they describe by means of preceding symbols. The possible preceding symbols are: no symbol, =, +, &, ?, and #.

- (no symbol)

  The operand is an input. It must appear as part of the third argument to the `asm()` construct. The allocated register is loaded with the value of the C/C++ expression before the `asm()` template is executed. Its C/C++ expression is not modified by the `asm()` construct, and its value may be a constant or literal.
  Example: `x`

- **= symbol**

  The operand is an output. It must appear as part of the second argument to the `asm()` construct. Once the `asm()` template has been executed, the value in the allocated register is stored into the location indicated by its C/C++ expression; therefore, the expression must be one that would be valid as the left-hand side of an assignment.
  Example: `=x`

- **+ symbol**

  The operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C/C++ expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C/C++ expression. Therefore, as with pure outputs, the C/C++ expression must be one that is valid on the left-hand side of an assignment.
  Example: `+x`

- **? symbol**

The operand is temporary. It must appear as part of the third argument to the `asm()` construct. A register is allocated as working space for the duration of the `asm()` template execution. The register's initial value is undefined, and the register's final value is discarded. The corresponding C/C++ expression is not loaded into the register, but must be present. This expression is normally specified using a literal zero.
Example: `?x`

- **& symbol**

  This operand constraint may be applied to inputs and outputs. It indicates that the register allocated to the input (or output) may not be one of the registers that are allocated to the outputs (or inputs). This operand constraint is used when one or more output registers are set while one or more inputs are yet to be referenced. (This situation sometimes occurs if the `asm()` template contains more than one instruction.)
  Example: `&x`

- **# symbol**

  The operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. An input operand with this constraint will not be allocated the same register as any other input or output operand of the `asm()`. The operand must appear as part of the second argument to the `asm()` construct.
  Example: "`#x`"

It is also possible to claim registers directly, instead of requesting a register from a certain class using the constraint letters. You can claim the registers directly by simply naming the register in the location where the class letter would be.

For example,

```
asm("%0 += FDEP %1 BY %2;;"
:"+XR0"(sum)      /* output */
:"x"(x),"x"(y)    /* input  */
);
```

would load `sum` into `XR0`, and load `x` and `y` into two X registers, execute the operation, and then store the new value from `XR0` back into `sum`.

Naming the registers in this way allows the `asm()` construct to specify several registers that must be related, such as the Base and Length registers for a circular buffer. This also allows use of resister not covered by the register classes accepted by the `asm()` construct.

The clobber string can be any of the registers recognized by the compiler. These should all be in upper case; for example, "`XR10`", "`YMR0`", "`XTR3`", or "`LC0`".

Table 1-18. Constraint Register Types

Constraint[1]	Register Type
`x`	Compute block X register
`y`	Compute block Y register
`j`	JALU register
`k`	KALU register

1 Names are case-sensitive.

Table 1-19. Constraint Operators

Constraint Operator[1]	Description
`constraint`	Indicates the constraint is an input operand
`constraintl`	Indicates the constraint is long (64bit) register
`constraintq`	Indicates the constraint is quad (128-bit) register

Table 1-19. Constraint Operators (Cont'd)

Constraint Operator[1]	Description
=constraint	Indicates the constraint is applied to an output operand
&constraint	Indicates the constraint is applied to an input operand that may not be overlapped with an output operand
=&constraint	Indicates the constraint is applied to an output operand that may not overlap an input operand
?constraint	Indicates the constraint is temporary
+constraint	Indicates the constraint is both an input and output operand
#constraint	Indicates the constraint is an input operand whose value are changed
constraintb	Indicates the constraint is is applied to a byte operation
constraints	Indicates the constraint is applied to a short operation.
constraintL	Indicates the constraint is applied to a long operation.
constraintf	Indicates the constraint is applied to a float operation.

1 Names are case-sensitive.

Note that if the `q` or `l` constraints are specified, then they must be specified before the constraints listed above. For example,

The following inline assembler statement,

```
asm("%0 = LSHIFT %1 BY -1;;" : "=xlL" (input) : "xl" (input) );
```

where input and output are defined as `long long` types. produces a core instruction similar to

```
XLR15:14 = LSHIFT R13:12 BY -1;;
```

Assembly Constructs With Multiple Instructions

There can be many assembly instructions in one template. The input operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like. If the `asm()`

string is longer than one line, you may continue it on the next line by placing a backslash (\) at the end of the line or by quoting each line separately.

This is an example of multiple instructions in a template:

```
/* (pseudo code) k2 = x; k3 = y; result = x + y; */
asm ("k2=%1;;  \
    k3=%2;;    \
    %0=k2+k3;;"
    : "=k" (result)            /* output   */
    : "k" (x), "k" (y)         /* input    */
    : "k2", "k3");             /* clobbers */
```

Do not attempt to produce multiple-instruction `asm` constructs via a sequence of single-instruction `asm` constructs, as the compiler is not guaranteed to maintain the ordering. For example, the following should be avoided:

```
/* BAD EXAMPLE: Do not use sequences of single-instruction
** asms. Use a single multiple-instruction asm instead.  */

asm("k2=%0;;" : : "k" (x) : "k2");
asm("k3=%0;;" : : "k" (y) : "k3");
asm("%0=k2+k3;;" : "=k" (result));
```

Assembly Construct Reordering and Optimization

For the purpose of optimization, the compiler assumes that the side effects of an `asm()` construct are limited to changes in the output operands. This assumption does not mean that you cannot use instructions with side effects, but you must be careful to notify the compiler that you are using them by using the clobber specifiers.

The compiler may eliminate them if the output operands are not used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

Use the keyword `volatile` to prevent an `asm()` instruction from being moved, combined, or deleted. For example,

```
#define GetCycleCount(counter)  \
asm volatile ("%0 = CCNT0;;" : "=j"(counter) );
```

A sequence of `asm volatile()` constructs is not guaranteed to be completely consecutive; it may be moved across jump instructions or in other ways that are not significant to the compiler. To force the compiler to keep the output consecutive, use only one `asm volatile()` construct, or use the output of the `asm()` construct in a C statement.

Assembly Constructs With Input and Output Operands

The assembly constructs' output operands must be write only; the compiler assumes that the values in these operands do not need to be preserved. When the assembler instruction has an operand that is both read from and written to, you must logically split its function into two separate operands: one input operand and one write-only output operand. The connection between them is expressed by constraints that say they need to be in the same location when the instruction executes.

When a register's value is both an input and an output, and the final value is to be stored to the same location from which the original value was loaded, the register can be marked as an input-output, using the "+" constraint symbol, as described earlier.

If the final value is to be saved into a different location, then both an input and an output must be specified, and the input must be tied to the output by using its position number as the constraint.

For example,

```
asm("[%0 +=2] = J4;;"
   :"=j" (newptr)      // an output, given a jreg,
                       // stored into newptr.
   :"0" (oldptr));     // an input, given same reg as %0,
                       // initialized from oldptr
   )
```

Assembly Constructs and Flow Control

Do not place flow control operations within an `asm()` construct that "leaves" the `asm()` construct functions, such as calling a procedure or performing a jump, to another piece of code that is not within the `asm()` construct itself. Such operations are invisible to the compiler, may result in multiple-defined symbols, and may violate assumptions made by the compiler.

For example, the compiler is careful to adhere to the calling conventions for preserved registers when making a procedure call. If an `asm()` construct calls a procedure, the `asm()` construct must also ensure that all conventions are obeyed, or the called procedure may corrupt the state used by the function containing the `asm()` construct.

It is also inadvisable to use labels in `asm()` statements, especially when function inlining is enabled. If a function containing such `asm` statements is inlined more than once in a file, there will be multiple definitions of the label, resulting in an assembler error. If possible, use PC-relative jumps in `asm` statements.

Guidelines on the Use of asm() Statements

There are certain operations that are performed more efficiently using other compiler features, and result in source code that is clearer and easier to read.

Accessing System Registers

System registers are accessed most efficiently using the functions in `sysreg.h` instead of using `asm()` statements.

64-Bit Integer Support (long long)

In addition to the basic C/C++ data types, the `ccts` compiler provides an additional integral type that consists of 64-bit integers. This type is provided in both signed and unsigned forms. It is fully supported, including all arithmetic operations and relevant libraries.

Table 1-20. Long long Datatype Characteristics

Datatype Characteristic	Description
Type Name	`long long [int]` `unsigned long long [int]`
Representation	64-bit 2's complement
Range	`long long int: -2**63 ... (2**63)-1` `unsigned long long int:  0 ... (2**64) -1`

Normal library functions (as in `<stdlib.h>`) taking this type typically have an "`ll`" prefix. Formatting support (`printf`, etc.) is available via the "`ll`" modifier, as in "`%lld`".

This extension does not introduce additional keywords, so it does not interfere with porting standard-conforming programs onto TigerSHARC processors. The extension is always available; the `-pedantic` switch does not affect it.

Quad-Word Support

Certain features of TigerSHARC processors make use of 128-bit data types (a quad word). An extension to the language provides a data type of this size. Normal C/C++ arithmetic operators are not supported for the quad word type; therefore, supplied built-in functions must be used to process variables of this type. The type name is `__builtin_quad`.

Memory Support Keywords (pm dm)

There are two keywords used to designate memory space: `dm` and `pm`. These keywords can be used to specify the location of a static or global variable or to qualify a pointer declaration.

These keywords allow you to control placement of data in primary (`dm`) or secondary (`pm`) data memory.

Memory Keyword Rules

These rules apply to memory support keywords:

- A memory space keyword (`dm` or `pm`) refers to the expression to its right.
- You can specify a memory space for each level of pointer. This corresponds to one memory space for each `*` in the declaration.
- The compiler uses Data Memory (`dm`) as the default memory space for all variables. All undeclared spaces for data are Data Memory spaces.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the stack, which is always in Data Memory.
- Literal character strings always reside in Data Memory.

The following listing shows examples of memory keyword syntax.

```
int pm abc[100];
/* declares an array abc with 100 elements in pm (secondary
   data memory)                                              */
int dm def[100];
/* declares an array def with 100 elements in primary data memory */
int ghi[100];
/* declares an array ghi with 100 elements in primary data memory */
int pm * pm pp;
/* declares pp to be a pointer which resides in secondary data
   memory and points to a pm (econdary data memory) integer    */
int dm * dm dd;
/* declares dd to be a pointer which resides in a primary data
   memory and points to a primary data memory integer          */
int *dd;
/* declares dd to be a pointer which points to a primary data
   memory integer                                             */
int pm * dm dp;
/* declares dp to be a pointer which resides in a primary data
   memory and points to a pm (secondary data memory) integer     */
int pm * dp;
/* declares dp to be a pointer which resides in a primary data
   memory and points to a pm (secondary data memory) integer    */
int dm * pm pd;
/* declares pd to be a pointer which resides in pm (secondary
   data memory) and points to a primary data memory integer    */
int * pm pd;
/* declares pd to be a pointer which resides in pm (secondary
   data memory) and points to a primary data memory integer    */
float pm * dm * pm fp;
/* the first pm means that *fp is in pm (secondary data memory,
   the following dm puts *fp in primary data memory, and fp
   itself is in pm (secondary data memory                      */
```

Memory space specification keywords cannot qualify type names and structure tags, but you can use them in pointer declarations. The following listing shows examples of memory space specification keywords in `typedef` and `struct` statements.

```
/* Dual Memory Support Keyword typedef & struct Examples */
   typedef float pm * PFLOATP;
          /* PFLOATP defines a type which is a pointer to a */
          /* float which resides in pm.                     */

   struct s {int x; int y; int z;};
   static pm struct s mystruct={10,9,8};
          /* Note that the pm specification is not used in  */
          /* the structure definition. The pm specification */
          /* is used when defining the variable mystruct    */
```

__regclass Construct

The `__regclass(unit)` construct allows you to give the compiler hints about which TigerSHARC processor unit to use for a particular computation. Variables can be annotated with this construct, and the compiler then tries to perform computations involving those variables in the indicated units. The `(unit)` argument is a quoted string, naming one of the four major units: “`X`”, “`Y`”, “`J`”, “`K`”.

The `__regclass` construct may be used anywhere that the C register qualifier may be used and is subject to the same restrictions: it cannot have its address taken. Effective use of this feature requires a good understanding of the TigerSHARC architecture. Typically, examine the assembly code produced by the optimizing compiler.

In the absence of specific `__regclass` hints, the TigerSHARC compiler uses the following criteria:

- Floating-point calculations are done in compute block X
- General integer calculations are done in compute block Y
- Address calculations, including pointer arithmetic, are done in the JALU. If a multiplication is needed, it is done in Y

A common way of achieving higher performance, particularly in tight loops, is to perform a part of the computation in a different unit. If the sub-computations assigned to different units are independent of each other, they execute in parallel, resulting in increased throughput.

Bank Type Qualifiers

Bank qualifiers can be attached to data declarations to indicate that the data resides in particular memory banks. For example,

```
int bank("blue") *ptr1;
int bank("green") *ptr2;
```

The bank qualifier assists the optimizer because the compiler assumes that if two data items are in different banks, they can be accessed together without conflict. The bank name string literals have no significance, except to differentiate between banks. There is no interpretation of the names attached to banks, which can be any arbitrary string. There is a current implementation limit of ten different banks.

For any given function, three banks are automatically defined. These are:

- The default bank for global data.
  The "static" or "extern" data that is not explicitly placed into another bank is assumed to be within this bank. Normally, this bank is called "`__data`", although a different bank can be selected with `#pragma data_bank(bankname)`.

- The default bank for local data.
  Local variables of "auto" storage class that are not explicitly placed into another bank are assumed to be within this bank. Normally, this bank is called "__stack", although a different bank can be selected with #pragma stack_bank(bankname).
- The default bank for the function's instructions.
  The function itself is placed into this bank. Normally, it is called "__code", although a different bank can be selected with #pragma code_bank(bankname).

Each memory bank can have different performance characteristics. For more information on memory bank attributes, see "Memory Bank Pragmas" on page 1-226.

Placement Support Keyword (section)

The section() keyword directs the compiler to place an object or function in an assembly .SECTION directive, in the compiler's intermediate assembly output file. Name the assembly .SECTION directive using the string literal parameter of the section() keyword. If you do not specify a section() for an object or function declaration, the compiler uses a default section. The .LDF file supplied to the linker must also be updated to support the additional named sections.

Applying the section() function is only meaningful when the data item is something that the compiler can place in the named section.

Apply the section() keyword only to top-level, named objects that have static duration; for example, they are explicitly static, or are given as external-object definitions. The example shows the declaration of a static variable that is placed in the section called bingo.

```
static section("bingo") int x;
```

Placement of Compiler-Generated Code and Data

If the `section()` keyword ("Placement Support Keyword (section)" on page 1-119) is not used, the compiler emits code and data into default sections. The `-section` switch (on page 1-60) can be used to specify alternatives for these defaults on the command-line, and the `default_section` pragma (on page 1-212) can be used to specify alternatives for some of them within the source file.

In addition, when using certain features of C/C++, the compiler may be required to produce internal data structures. The `section` switch and the `default_section` pragma allow you to override the default location where the data would be placed. For example,

```
ccts -section vtbl=vtbl_data test.cpp -c++
```

would instruct the compiler to place all the C++ virtual function look-up tables into the section `vtbl_data`, rather than the default `vtbl` section. It is the user's responsibility to ensure that appropriately named sections exist in the `.ldf` file.

The following section identifiers are currently supported:

`code` – controls placement of machine instructions. Default is `program`.

`data` – controls placement of initialized variable data. Default is `data1`.

`bsz` – controls placement of zero-initialized variable data. Default is `bsz`.

`sti` – controls placement of the static C++ class constructor functions. Default is `program`.

`vtbl` – controls placement of the C++ virtual lookup tables. Default is `vtbl`.

`vtable` – synonym for `vtbl`

`switch` – controls placement of switch-statement jump-tables. Default is `data1`.

When both `-section` switches and `default_section` pragmas are used, the following priority is used:

1. A `default_section` pragma within the source has the highest priority.
2. The `-section` switch has precedence if no `default_section` pragma is in force.

Boolean Type Support Keywords

The `bool`, `true`, and `false` keywords are extensions to ANSI C that support the C++ Boolean type. The `bool` keyword is a unique signed integral type. There are two built-in constants of this type: `true` and `false`. When converting a numeric or pointer value to `bool`, a zero value becomes `false`; a nonzero value becomes `true`. A `bool` value may be converted to `int` by promotion, taking `true` to one and `false` to zero. A numeric or pointer value is automatically converted to `bool` when needed.

These keywords behave more or less as if the declaration that follows had appeared at the beginning of the file, except that assigning a nonzero integer to a `bool` type always causes it to take on the value `true`.

```
typedef enum { false, true } bool;
```

Pointer Class Support Keyword (restrict)

The `restrict` keyword is an extension that supports restricted pointer features. The use of `restrict` is limited to the declaration of a pointer. This keyword specifies that the pointer provides exclusive initial access to the pointed object. More simply, the `restrict` keyword is a way to identify that a pointer does not create an alias. Also, two different restricted pointers cannot designate the same object and therefore they are not aliases.

The compiler is free to use the information about restricted pointers and aliasing in order to better optimize C or C++ code that uses pointers. The `restrict` keyword is most useful when applied to function parameters that the compiler otherwise have little information about.

The behavior of a program is undefined if it contains an assignment between two restricted pointers except for the following cases:

- A function with a restricted pointer parameter may be called with an argument that is a restricted pointer.
- A function may return the value of a restricted pointer that is local to the function, and the return value may then be assigned to another restricted pointer.

If you have a program that uses a restricted pointer in a way that it does not uniquely refer to storage, then the behavior of the program is undefined.

Variable-Length Array Support

The compiler supports variable-length automatic arrays. Unlike other automatic arrays, variable-length ones are declared with a non-constant length. This means that the space is allocated when the array is declared, and deallocated when the brace-level is exited.

Variable-length arrays are only supported as an extension to C and not C++.

The compiler does not allow jumping into the brace-level of the array and produces a compile time error message if this is attempted. The compiler does allow breaking or jumping out of the brace-level, and it deallocates the array when this occurs.

You can use variable-length arrays as function arguments, such as:

```
struct entry
   var_array (int array_len, char data[array_len][array_len])
   {
   ...
   }
```

The compiler calculates the length of an array at the time of allocation. It then remembers the array length until the brace-level is exited and can return it as the result of the `sizeof()` function performed on the array.

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as input matrices. Declaring automatic variable size matrix is much easier then explicitly allocating it in a heap.

The expression declares an array with a size that is computed at run time. The length of the array is computed on entry to the block and saved in case `sizeof()` is applied to the array. For multidimensional arrays, the boundaries are also saved for address computation. After leaving the block, all the space allocated for the array and size information is deallocated.

For example, the following program prints 40, not 50:

```
#include <stdio.h>
void foo(int);

main ()
{
   foo(40);
}

void foo (int n)
()
{
   char c[n];
   n = 50;
```

```
    printf("%d", sizeof(c));
}
```

Non-Constant Aggregate Initializer Support

The ccts compiler includes support for the ISO/ANSI standard definition of C and C++ and also includes extended support for initializers. The compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions.

The following example shows an initializer with elements that vary at run time:

```
void initializer (float a, float b)
{
   float the_array[2] = { a-b, a+b };
}

void foo (float f, float g)
{
   float beat_freqs[2] = { f-g, f+g };
}
```

Indexed Initializer Support

The ISO/ANSI Standard C and C++ requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized. The ccts compiler, by comparison, supports labeling elements for array initializers. This feature lets you specify array or structure elements in any order by specifying the array indices or structure field names to which they apply. All index values must be constant expressions, even in automatic arrays.

For an array initializer, the syntax `[INDEX]` appearing before an initializer element value specifies the index to be initialized by that value. Subsequent initializer elements are then applied to the sequentially following

elements of the array, unless another use of the [INDEX] syntax appears. The index values must be constant expressions, even if the array being initialized is automatic.

The following example shows equivalent array initializers—the first initializer is in ISO/ANSI standard C/C++; the second initializer uses the ccts compiler.

The [index] precedes the value being assigned to that element.

```
/* Example 1 Standard & ccts C/C++ Array Initializer */
/* Standard Array Initializer */

int a[6] = { 0, 0, 115, 0, 29, 0 };

/* equivalent ccts C/C++ array initializer */

int a[6] = { [2] 115, [4] 29 };
```

You can combine this technique of naming elements with Standard C/C++ initialization of successive elements. The standard and ccts instructions below are equivalent. Note that any unlabeled initial value is assigned to the next consecutive element of the structure or array.

```
/* Example 2 Standard & ccts C/C++ Array Initializer */
/* Standard Array Initializer */

int a[6] = { 0, v1, v2, 0, v4, 0 };

/* equivalent ccts C/C++ array initializer that uses
   indexed elements */

int a[6] = { [1] v1, v2, [4] v4 };
```

The following example shows how to label the array initializer elements when the indices are characters or enum type.

```
/* Example 3 Array Initializer With enum Type Indices */
/* ccts C/C++ array initializer */

int whitespace[256] =
```

```
{
   [' '] 1, ['\t'] 1, ['\v'] 1, ['\f'] 1, ['\n'] 1, ['\r'] 1
};
```

In a structure initializer, specify the name of a field to initialize with *field name* before the element value. The standard C/C++ and `ccts` C/C++ struct initializers in the example below are equivalent.

```
/* Example 4 Standard & ccts C/C++ struct Initializer */
/* Standard struct Initializer */

struct point {int x, y;};
struct point p = {xvalue, yvalue};

/* Equivalent ccts C/C++ struct Initializer With
Labeled Elements */

struct point {int x, y;};
struct point p = {y: yvalue, x: xvalue};
```

Compiler Built-In Functions

The compiler supports intrinsic (built-in) functions that enable you to make efficient use of hardware resources. The compiler is aware of the definition of these intrinsic functions without the use of a header file. Your program uses them via normal function call syntax. The compiler notices the invocation and generates one or more machine instructions, just as it does for normal operators, such as '+' or '*'.

Built-in functions have names that begin with `__builtin`.

Identifiers beginning with '__' are reserved by the C standard, so these names do not conflict with user-defined identifiers.

This section describes:

- "Using the builtins.h Header File" on page 1-127
- "Optimization Guidance Built-in Functions" on page 1-129

- "16-Bit Data Types" on page 1-131
- "32-Bit Data Types" on page 1-140
- "Circular Buffer Built-In Functions" on page 1-142
- "Math Intrinsics" on page 1-143
- "Instructions Generated by Built-in Functions" on page 1-145
- "Data Alignment Buffer (DAB) Built-in Functions" on page 1-161
- "Circular Buffer Data Alignment Buffer (DAB) Built-in Functions" on page 1-162
- "Communications Logic Unit Operations" on page 1-164

These functions are specific to individual architectures; the built-in functions supported on TigerSHARC processors are described in subsections that follow.

Various system header files provide the user with definitions and access to the intrinsics. This feature may be disabled using the `-no-builtin` switch. (See on page 1-45.)

Using the builtins.h Header File

The `builtins.h` header file provides user-visible prototypes for the built-in functions, though as noted above, these are not necessary for the compiler as it already has the information about the built-in prototypes. It does, however, also provide shorthand forms of some of the more complicated built-in functions (and it is strongly recommended that these are used over their constituent built-in functions).

The `builtins.h` header file also provides C reference code for the built-in functions. The header file can thus be included in compiler for architectures other than the TigerSHARC processors and can enable compilation

and execution of code that makes use of the TigerSHARC built-in functions. Examining the C reference code can provide insight into the function that a given built-in function performs.

When using a compiler other than the one provided with VisualDSP++ for TigerSHARC processors, please make sure your compiler supports inline functions as the C reference versions of the built-ins use the `inline` keyword. These implementations of the built-in functions also require support for 64-bit integers.

When using this file on a machine that does not support byte-addressing, define `__NO_BYTE_ADDRESSING__` before including this file. If the machine has 8-bit `char` types and 16-bit `short` types, then do not define this macro.

To use the reference implementation of the `XCORRS` Communication Logic Unit instruction in projects being built for the ADSP-TS101 processor, define the macro `__USE_RAW_XCORRS__` before including the `builtins.h` file.

When using the Microsoft C/C++ compiler, use the `/TP` command-line option to select C++ compilation: "`inline`" is not recognized as a keyword in C compilation mode.

To allow your compilation to ignore all of the `__builtin_sysreg_read` and `__builtin_sysreg_write` built-in functions, define `__IGNORE_SYSREG_BUILTINS__` before including this file allowing the code to compile. To allow your compilation to ignore the `__builtin_idle` built-in functions, define `__IGNORE_IDLE_BUILTINS__` before including this file.

To use the raw versions of the built-in functions even when using the compiler provided with VisualDSP++ for TigerSHARC processors, define `__USE_RAW_BUILTINS__` before including this file.

Optimization Guidance Built-in Functions

This section describes the optimization guidance built-in functions.

```
void __builtin_aligned(const void *p, int align);
```

The `__builtin_aligned` intrinsic is an assertion that the first parameter points to an argument that is aligned on a multiple of the second argument. The second argument is in units of `char` size, that is 8-bit bytes in byte-addressing mode and 32-bit words in word-addressing mode. Knowing alignment helps the optimizer to combine loads and stores from successive iterations of a loop. Ideally, all data processed in the first iteration of the loop should be aligned on a 4-word boundary.

An example in word-addressing mode where the parameter "`a`" points to quad-word aligned data would be as follows:

```
void fn(int *a) {
   __builtin_aligned(a, 4);
....
}
```

The compiler performance built-in functions provide the compiler with information about the expected behavior of the program. You can use these built-in functions to tell the compiler which parts of the program are most likely to be executed; the compiler can then arrange for the most common cases to be those that execute most efficiently.

```
#include <builtins.h>
int __builtin_expected_true(int cond);
int __builtin_expected_false(int cond);
```

For example, consider the code

```
extern int func(int);
int example(int call_the_function, int value)
{
   int r = 0;
   if (call_the_function)
```

```
        r = func(value);
    return r;
}
```

If you expect that parameter `call_the_function` to be true in the majority of cases, you can write the function in the following manner:

```
extern int func(int);
int example(int call_the_function, int value)
{
    int r = 0;
    if (__builtin_expected_true(call_the_function))
                    // indicate most likely true
        r = func(value);
    return r;
}
```

This indicates to the compiler that you expect `call_the_function` to be true in most cases, so the compiler arranges for the default case to be to call function `func()`. If, on the other hand, you were to write the function as:

```
extern int func(int);
int example(int call_the_function, int value)
{
    int r = 0;
    if (__builtin_expected_false(call_the_function))
                     // indicate most likely false
        r = func(value);
    return r;
}
```

then the compiler arranges for the generated code to default to the opposite case, of not calling function `func()`.

These built-in functions do not change the operation of the generated code, which will still evaluate the boolean expression as normal. Instead, they indicate to the compiler which flow of control is most likely, helping the compiler to ensure that the most commonly-executed path is the one that uses the most efficient instruction sequence.

The __builtin_expected_true and __builtin_expected_false built-in functions only take effect when optimization is enabled in the compiler. They are only supported in conditional expressions.

16-Bit Data Types

The functions and operators described in this section perform arithmetic on the int2x16, and int4x16 data types.

- The int2x16 data type consists of two 16-bit integers packed into a 32-bit structure.
- The int4x16 data type consists of four 16-bit integers packed into a 64-bit item.

In the following diagram, the notation {A,B} represents an int2x16 containing the two 16-bit values A and B. The TigerSHARC processors have a little endian architecture, so the first element is stored in the low order bits. A is considered to be the first element of {A,B}, so that the representation in a 32-bit register (int2x16 data type) has the following structure.

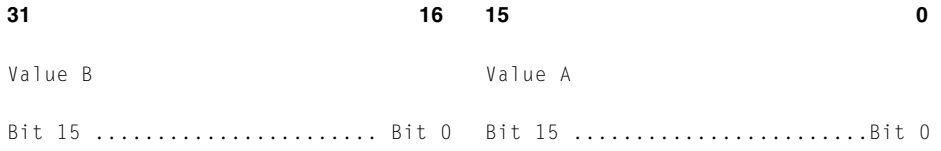

Figure 1-2. 16-Bit Data Type

Similarly, {A,B,C,D} represent an int4x16 containing the 16-bit values A, B, C, and D, with A in the lowest order bits, B next, followed by C, and D in the highest order bits. (This means that C and D are stored in a register or memory location numbered one higher than A and B.)

Most operations on these types are element-wise. For example, the add operation for int2x16 takes the first 16-bit quantity from each argument and adds them together to produce the first 16-bit quantity in the result.

Likewise, the second 16-bit elements are added to produce the second element of the result. For example, the `int2x16` add operation on arguments `{A,B}` and `{C,D}` produces a result `{A+C,B+D}`.

A few operations which are not element wise are also supported. The sideways sum operation adds up all of the 16-bit elements in its single argument. A variety of packing and unpacking operations for accessing the individual elements of a packed value are also available.

These operations are very efficient for the ADSP-TSxxx processors because they are directly supported in hardware. Most are implemented as a single machine instruction; a few which can take multiple machine instructions are provided for convenience. Note that some operations are defined in terms of a `2x32` data type. This type and its operations are described in "32-Bit Data Types" on page 1-140.

Packed 16-bit Integer Support Using C

The C language version used at Analog Devices lets you write your own program to operate on 16-bit packed data. To use the `int2x16` and `int4x16` types and operators in your Analog Devices C program, add a `#include <i16.h>` directive.

Constructors (int2x16 values)

The following functions create `int2x16` values.

Synopsis:

```
int2x16 compact_to_i2x16_from_i32 (int lo, int hi);
```

Description: Constructs an `int2x16` type by compacting two 32-bit integer values. The high-order 16 bits of each original 32-bit value are lost.

Algorithm:

```
compact(A_32, B_32) => {A_16,B_16}
```

Synopsis:

```
int2x16 compact_to_i2x16 (int2x32 val);
```

Description: Construct an `int2x16` type by compacting the two halves of a single `int2x32`. The high-order 16 bits of each of the 32-bit values are lost.

Algorithm:

```
compact({A_32,B_32}) => {A_16,B_16}
```

Extractors and Expanders (int2x16 values)

The following functions extract the latest or most significant 16-bit value from an `int2x16`.

Synopsis:

```
int expand_low_of_i2x16  (int2x16 val);
int expand_high_of_i2x16 (int2x16 val);
```

Description: The value returned is sign-extended to 32 bits and returned as an `int`.

Algorithm:

```
val = {A_16,B_16}:
expand_low_of_i2x16(val)  => A_32
expand_high_of_i2x16(val) => B_32
```

See Also: `expand_i2x16_to_i2x32`

Arithmetic Operators

The following binary operators are defined on `int2x16`:

Synopsis:

```
int2x16 add_i2x16  (int2x16 a, int2x16 b);
int2x16 sub_i2x16  (int2x16 a, int2x16 b);
int2x16 mult_i2x16 (int2x16 a, int2x16 b);
```

Description: Performs element-wise arithmetic on `int2x16` values. The TigerSHARC processors do not have a `2x16` multiplication instruction; this multiply operation uses the `4x16` multiply and discards half of the result. It is still accomplished in one instruction.

Algorithm:

Addition:	`{A,B} + {X,Y} => {A+X, B+Y}`
Subtraction:	`{A,B} - {X,Y} => {A-X, B-Y}`
Multiplication:	`{A,B} * {X,Y} => {A*X, B*Y}`

Unary minus is not yet supported on `int2x16`.

Bitwise Operators (int2x16 values)

The following bitwise logical operators are defined on `int2x16`:

Synopsis:

```
int2x16 a, b;
a ^ b
a & b
a | b
a ^= b
a &= b
a |= b
~a
```

Description: Performs logical operations on `int2x16` values. Note that the traditional `infix` operators are available here.

Algorithm:

Xor:	`{A,B} ^ {X,Y} => {A^X,B^Y}`
And:	`{A,B} & {X,Y} => {A&X,B&Y}`
Or:	`{A,B} | {X,Y} => {A|X,B|Y}`
Complement:	`{A,B} => {~A,~B}`

Comparison Operators (int2x16 values)

The following operators are used to compare `int2x16` values.

Synopsis:

```
int2x16 a, b;
a == b;
a != b;
```

Description: The == operator returns true if both elements are equal. The != operator returns true unless both values are equal. Other comparison operators (<, <=, >=, >) are not supported.

Algorithm:

Equal: `{A,B} == {C,D} => (A == B && C == D)`

Not Equal: `{A,B} != {C,D} => (A != B || C != D)`

Sideways Sum (int2x16 values)

This function adds together the two values in an `int2x16`.

Synopsis:

```
int sum_i2x16 (int2x16);
```

Description: This operation produces a single 32-bit integer result.

Algorithm:

```
sum({A,B}) => A+B
```

Example (int2x16 values)

This example shows some of the `int2x16` operations, including conversion to and from normal ints. Note that it is often helpful in debugging to print the packed values in hex, rather than decimal, as the two halves can be seen more easily.

```
#include <stdio.h>      // for printing at end
#include <i16.h>
void i2x16_example() {
   int u, v, s;
   int
      x = 3,
      y = 5,
      z = 7,
      w = 9;
   int2x16 aa, bb, cc, dd;
               // compact integers into packed 16 bit form
   aa = compact_to_i2x16_from_i32(x, y);
               // aa = {3, 5} or 0x00050003
   bb = compact_to_i2x16_from_i32(z, w);
               // bb = {7, 9} or 0x00090007
               // construct a packed constant
   cc = compact_to_i2x16_from_i32(1, -1);
               // cc = {1, -1} or 0xffff0001
               // a little arithmetic
   dd = mult_i2x16 (add_i2x16 (aa, bb), cc);
               // dd = (aa + bb) * cc;
               // dd = {10, -14} or 0xfff2000a
               // sideways sum
   s = sum_i2x16(dd);             //s = -4 or 0xfffffffc
                                  // extract components
   u = expand_low_of_i2x16(dd);   // low-order part
   v = expand_high_of_i2x16(dd);  // high-order part
   printf("results: s = %d; dd = %8x, u = %d, v = %d\n", s, dd, u, v);
}

 Prints:
 results: s = -4; dd = fff2000a, u = 10, v = -14
```

Constructors (int4x16 values)

The following functions create `int4x16` values

Synopsis:

```
int4x16 compact_to_i4x16_from_i32 (int llo, int lhi,
        int hlo, int hhi);
```

Description: Constructs an `int4x16` by compacting four 32-bit integer values. The high-order 16 bits of each original 32-bit value are lost.

Algorithm:

```
compact(A_32, B_32, C_32, D_32) => {A_16,B_16,C_16,D_16}
```

Synopsis:

```
int4x16 compose_i4x16_from_i2x16(int2x16 low, int2x16 high);
```

Description: Construct an `int4x16` from two `int2x16s`.

Algorithm:

```
compose ({A, B}, {C, D}) => {A,B,C,D}
```

Extractors (2x16 from a 4x16 value)

This function extracts a `2x16` from a `4x16`

Synopsis:

```
int2x16 extract_low_of_i4x16(int4x16 val);
int2x16 extract_high_of_i4x16(int4x16 val);
```

Description: Extracts a `int2x16` from a `int4x16`. It is often convenient to use the various expand operators to break the `int4x16` apart in steps, holding onto the intermediate forms, as shown in the following listing.

Given:

```
int4x16 val = {A_16,B_16,C_16,D_16}:int2x16 low_part =
               extract_low_of_i4x16(val);
int2x16 high_part = extract_high_of_i4x16(val);
int a = expand_low_of_i2x16(low_part);
int b = expand_high_of_i2x16(low_part);
int c = expand_low_of_i2x16(high_part);
int d = expand_high_of_i2x16(high_part);
```

- or -

```
int4x16 val = {A_16,B_16,C_16,D_16}:
int2x32 low_part = expand_i2x16(extract_low_of_i4x16(val));
int2x32 high_part = expand_i2x16(extract_high_of_i4x16(val));
int a = low_32(low_part);
int b = high_32(low_part);
int c = low_32(high_part);
int d = high_32(high_part);
```

Split an `int4x16` into component integer values

Algorithm:

```
extract_low ({A_16,B_16,C_16,D_16}) => {A_16,B_16}
extract_high({A_16,B_16,C_16,D_16}) => {C_16,D_16}
```

Synopsis:

```
void expand_i4x16_to_i32(int4x16 input,
int *llo, int *lhi, int *hlo, int *hhi);
```

Description: Using a cascade of expansion and selection, this breaks a `4x16` apart into four separate integers.

Algorithm:

```
val = {A_16,B_16,C_16,D_16}:
int a, b, c, d;
expand_i4x16_to_i32 (val, &a, &b, &c, &d);
a => A_32
b => B_32
c => C_32
d => D_32
```

Arithmetic Operators (int4x16 values)

The following binary operators are defined on int4x16:

Synopsis:

```
int4x16 add_i4x16   (int4x16 a, int4x16 b);
```

```
int4x16 sub_i4x16   (int4x16 a, int4x16 b);
int4x16 mult_i4x16  (int4x16 a, int4x16 b);
```

Algorithm:

Addition:	`A,B,C,D} + {W,X,Y,Z} => {A+W, B+X, C+Y, D+Z}`
Subtraction:	`A,B,C,D} - {W,X,Y,Z} => {A-W, B-X, C-Y, D-Z}`
Multiplication:	`A,B,C,D} * {W,X,Y,Z} => {A*W, B*X, C*Y, D*Z}`

Description: Performs element wise arithmetic on int4x16 values. Unary minus is not yet supported on `int4x16`.

Sideways Sum (int4x16 values)

This function adds together the four values in an `int4x16`.

Synopsis:

```
int sum_i4x16(int4x16);
```

Description: This operation produces a single 32-bit integer result.

Algorithm:

```
sum({A,B,C,D}) => A+B+C+D
```

Example (int4x16 values)

The following shows some examples of `int4x16` operations, including conversion to and from normal `ints`.

```
#include <stdio.h>
#include <i16.h>
void i4x16_example() {
   int w, x, y, z, s;
   int
      p = 3,
      q = 5,
      r = 7,
      s = 11,
      t = 13,
```

```
        u = 17,
        v = 19,
        g = 23;
    int4x16 aaaa, bbbb, cccc, dddd;
    // compact integers into packed 16 bit form
    aaaa = compact_to_i4x16_from_i32(p, q, r, s);
    bbbb = compact_to_i4x16_from_i32(t, u, v, g);
    cccc = compact_to_i4x16_from_i32(1, -1, 3, -3);
                    // construct a packed constant
                    // a little arithmetic
    dddd = mult_i4x16 (add_i4x16 (aaaa, bbbb), cccc);
                    // dddd = (aaaa + bbbb) * cccc;
    s = sum_i4x16(dddd);   // sideways sum
                    // extract components
    w = dddd[0];   // low-order part
    x = dddd[1];   // high-order part
    y = dddd[2];
    z = dddd[3];
    printf("results: s = %d; w = %d, x = %d, y = %d, z = %d\n",
        w, x, y, z);
}

Prints:
results: sm = -30; w = 16, x = -22, y = 78, z = -102
```

32-Bit Data Types

The `int2x32` data type is primarily used internally when converting between packed and expanded values. It often provides a useful way-station, allowing you to capture a partial expansion for later separation into smaller parts.

Constructors (int2x32 values)

The following functions create `int2x32` values.

Synopsis:

```
int2x32 compose_64 (int lo, int hi);
```

Algorithm:

```
compose(A_32, B_32) => {A_32, B_32}
```

Description: Constructs an `int2x32` from two 32-bit integer values. The values are unchanged.

Synopsis:

```
int2x32 expand_i2x16 (int2x16 val)
```

Algorithm:

```
expand ({A_16,B_16}) => {A_32, B_32}
```

Description: Expands an `int2x16` to an `int2x32` sign extending each value to 32 bits.

Extractors (int2x32 values)

The following functions extract the least or most significant 32-bit value from an `int2x32`. The unchanged 32-bit value is returned as an `int`.

Synopsis:

```
int low_32 (int2x32 val);
int high_32 (int2x32 val);
```

Algorithm:

```
low(A_32, B_32) => A_32
high(A_32, B_32) => B_32
```

Circular Buffer Built-In Functions

The C/C++ compiler provides the following two built-in functions for using the TigerSHARC circular buffer mechanisms.

If the compiler is being used in byte addressing mode, the circular buffer support only works for types of size `int` and greater. With the current compiler version, use of `short`, `char` or `void` pointers could result in incorrect run-time behavior.

Circular Buffer Increment of an Index

The following operation performs a circular buffer increment of an index.

```
int __builtin_circindex(int index ,int incr ,int nitems )
```

The operation is:

```
index +=incr;
if (index <0) index += nitems;
else if (index >= nitems) index -=nitems;
```

Circular Buffer Increment of a Pointer

The following operation

```
void *__builtin_circptr(void *ptr ,size_t incr,
                        void *base ,size_t buflen)
```

performs a circular buffer increment of a pointer. Both `incr` and `buflen` are specified in bytes, since the operation deals in void pointers. The operation is:

```
ptr += incr;
if (ptr <base) ptr += buflen;
else if (ptr >= (base+buflen)) ptr -= buflen;
```

The result of this operation is the updated circular buffer pointer.

The compiler also attempts to generate circular buffer increments for modulus array references, such as `array[index %nitems]`. For this to happen, the compiler must be able to determine that the starting value for index is within the range `0..(nitems-1)`.

When the "-force-circbuf" switch (on page 1-35) is specified, the compiler always treats array references of the form "`[i%n]`" as a circular buffer operation on the array.

Math Intrinsics

Each of the functions below map directly to a single machine instruction and are therefore very efficient. When using these functions, the compiler substitutes machine instructions unless you specify the command line "-no-builtin" option (on page 1-45).

```
/* Intrinsics defined in <stdlib.h> */
      /* "int" versions */
int   abs  (int j);               // absolute value
int   avg  (int a, int b);        // (a+b)/2
int   clip (int a, int b);
      // bound a by b (positive and negative)
int   max  (int a, int b);        // maximum
int   min  (int a, int b);        // minimum
int   count_ones (int numb);      // number of "1" bits

      /* "long int" versions */
long  labs  (long j);
long  lavg  (long a, long b);
long  lclip (long a, long b);
long  lmax  (long a, long b);
long  lmin  (long a, long b);

int   lcount_ones (long numb);

      /* "long long int" versions */
long long int   llabs (long long int j);
long long int   llavg  (long long int a, long long int b);
long long int   llclip (long long int a, long long int b);
```

```
long long int   llmax  (long long int a, long long int b);
long long int   llmin  (long long int a, long long int b);

int             llcount_ones (long long int numb);

/* bit-reversed addition */
int   addbitrev (int input_address, int number_of_bits);

/* Math intrinsics from <math.h>  */
float   fabsf (float, float);
float   favgf (float a, float b);   // (a+b)/2
float   fclipf (float a, float b);  // bound a by b (pos & neg)
float   fmaxf (float, float);          // float maximum
float   fminf (float, float);          // float minimum
float   copysignf (float a, float b);  // a with sign(b)
float signf (float a, float b);        // a with sign(b)

double   favgd (double, double);
double   fclipd (double, double);
double   fmaxd (double, double);
double   fmind (double, double);
double   copysignd (double, double);

long double   favgd (long double, long double);
long double   fclipd (long double, long double);
long double   fmaxd (long double, long double);
long double   fmind (long double, long double);
long double   copysignd (long double, long double);
```

RECIPS

Instruction:

```
Rs = RECIPS Rm
```

Builtins:

```
float __builtin_recip (float);
```

Example:

```
float r, a;
r = __builtin_recip (a);
```

Description:

`r` corresponds to `Rsd`

`a` corresponds to `Rmd`

RSQRTS

Instruction:

```
Rs = RSQRTS Rm
```

Builtins:

```
float __builtin_rsqrt (float);
```

Example:

```
float r, a;
r = __builtin_rsqrt (a);
```

Description:

`r` corresponds to `Rsd`

`a` corresponds to `Rmd`

Instructions Generated by Built-in Functions

The `ccts` compiler supports access to machine-specific instructions via built-in functions. These functions are known to the compiler without the need to declare them (via a header file). However, for convenience, a header file is included with the other C `include` functions (`builtins.h`). This header file gives prototypes for all the built-in functions. This file is quite terse, but it does provide the correct call syntax for each function.

Every function name is prefixed with `__builtin_` followed by a functional name, which usually corresponds directly to the machine instruction that the built-in function generates. Some functions, such as `compose` or `extract`, may not generate code under optimization since the compiler may be able to arrange register assignments to eliminate the need for these functions.

The following are listings of the built-in functions currently supported by the compiler and the instructions that they generate. Note that the listed instructions are the ones the compiler uses by default. As part of the optimization process, the compiler may replace instruction sequences with alternate equivalent sequences.

Many of the built-in functions map to a single machine instruction. The first section of this list comprises these built-in functions. Further down in the list are instructions which do not correspond directly to a machine instruction but which can be implemented simply by other means. (For example, extracting the low half of a 64-bit item can be done by the compiler making use of the low half of the item and ignoring the upper half.) More complex built-in functions are dealt with in separate sections based on their function.

For details on a specific function, please refer to the instruction set reference, the programming reference, or the hardware reference for the appropriate TigerSHARC target processor.

Addition and Subtraction:

```
int __builtin_add_sat(int Rm_1, int Rm_2);
   Rs = Rm_1 + Rm_2 (S);;
int __builtin_sub_sat(int Rm_1, int Rm_2);
   Rs = Rm_1 - Rm_2 (S);;
unsigned int __builtin_uadd_sat(unsigned int Rm_1, unsigned int Rm_2);
   Rs = Rm_1 + Rm_2 (SU);;
unsigned int __builtin_usub_sat(unsigned Rm_1, unsigned int Rm_2);
   Rs = Rm_1 - Rm_2 (SU);;
int __builtin_add_4x8(intRm, int Rn);
   BRs = Rm + Rn;;
int __builtin_add_4x8_sat(intRm, int Rn);
   BRs = Rm + Rn (S);;
int __builtin_sub_4x8(intRm, int Rn);
   BRs = Rm - Rn;;
int __builtin_sub_4x8_sat(intRm, int Rn);
   BRs = Rm - Rn (S);;
int __builtin_add_2x16(intRm, int Rn);
```

```
   SRs = Rm + Rn;;
int __builtin_add_2x16_sat(intRm, int Rn);
   SRs = Rm + Rn (S);;
int __builtin_sub_2x16(intRm, int Rn);
   SRs = Rm - Rn;;
int __builtin_sub_2x16_sat(intRm, int Rn);
   SRs = Rm - Rn (S);;

unsigned int __builtin_add_u2x16(unsigned int Rm, unsigned int Rn);
   SRs = Rm + Rn;;
unsigned int __builtin_add_u2x16_sat(unsigned int Rm, unsigned int Rn);
   SRs = Rm + Rn (SU);;
unsigned int __builtin_sub_u2x16(unsigned int Rm, unsigned int Rn);
   SRs = Rm - Rn;;
unsigned int __builtin_sub_u2x16_sat(unsigned int Rm, unsigned int Rn);
   SRs = Rm - Rn (SU);;

long long int __builtin_add_8x8(long long Rmd, long long int Rnd);
   BRsd = Rmd + Rnd;;
long long int __builtin_add_8x8_sat(long long int Rmd,
                                    long long int Rnd);
   BRsd = Rmd + Rnd (S);;
long long int __builtin_sub_8x8(long long int Rmd, long long int Rnd);
   BRsd = Rmd - Rnd;;
long long int __builtin_sub_8x8_sat(long long int Rmd,
                                    long long int Rnd);
   BRsd = Rmd - Rnd (S);;
long long int __builtin_add_4x16(long long int Rmd, long long int Rnd);
   SRsd = Rmd + Rnd;;
long long int __builtin_add_4x16_sat(long long int Rmd,
                                     long long int Rnd);
   SRsd = Rmd + Rnd (S);;
long long int __builtin_sub_4x16(long long int Rmd, long long int Rnd);
   SRsd = Rmd - Rnd;;
long long int __builtin_sub_4x16_sat(long long int Rmd, long long int Rnd);
   SRsd = Rmd - Rnd (S);;

unsigned long long int __builtin_add_u4x16_sat(
                    unsigned long long int Rmd,
                    unsigned long long int Rnd);
```

```
   SRsd = Rmd + Rnd (SU);;
unsigned long long int __builtin_sub_u4x16_sat(
                    unsigned long long int Rmd,
                    unsigned long long int Rnd);
   SRsd = Rmd - Rnd (SU);;
long long int __builtin_add_2x32(long long int Rmd, long long int Rnd);
   Rsd = Rmd + Rnd;;
long long int __builtin_add_2x32_sat(long long int Rmd,
                                     long long int Rnd);
   Rsd = Rmd + Rnd (S);;
long long int __builtin_sub_2x32(long long int Rmd, long long int Rnd);
   Rsd = Rmd - Rnd;;
long long int __builtin_sub_2x32_sat(long long int Rmd,
                                     long long int Rnd);
   Rsd = Rmd - Rnd (S);;
unsigned long long int __builtin_add_2x32u(
                    unsigned long long int Rmd,
                    unsigned long long int Rnd);
   Rsd = Rmd + Rnd;;
unsigned long long int __builtin_add_u2x32_sat(
                    unsigned long long int Rmd,
                    unsigned long long int Rnd);
   Rsd = Rmd + Rnd (SU);;
unsigned long long int __builtin_sub_2x32u(
                    unsigned long long int Rmd,
                    unsigned long long int Rnd);
   Rsd = Rmd - Rnd;;
unsigned long long int __builtin_sub_u2x32_sat(
                    unsigned long long int Rmd,
                    unsigned long long int Rnd);
   Rsd = Rmd - Rnd (SU);;

__builtin_quad __builtin_add_4x32(__builtin_quad,
                                  __builtin_quad);
  two separate Rsd = Rmd + Rnd;;
__builtin_quad __builtin_add_4x32_sat(__builtin_quad,
                                      __builtin_quad);
  two separate Rsd = Rmd + Rnd (S);;
__builtin_quad __builtin_add_u4x32_sat(__builtin_quad,
                                       __builtin_quad);
```

```
  two separate Rsd = Rmd + Rnd (SU);;
__builtin_quad __builtin_sub_4x32(__builtin_quad,
                                  __builtin_quad);
  two separate Rsd = Rmd - Rnd;
__builtin_quad __builtin_sub_4x32_sat(__builtin_quad,
                                      __builtin_quad);
  two separate Rsd = Rmd - Rnd (S);;
__builtin_quad __builtin_sub_u4x32_sat(__builtin_quad,
                                       __builtin_quad);
               two separate Rsd = Rmd - Rnd (SU);;
int __builtin_addbitrev(intJm, int Jn);
   Js = Jm + Jn (BR);;
```

Conversion:

```
int __builtin_compact_to_fr2x16(long long int Rmd);
   SRs = COMPACT Rmd;;
int __builtin_compact_to_fr2x16_trunc(long long int);
   SRs = COMPACT Rmd (T);;
int __builtin_compact_to_i2x16_sat(long long int Rmd);
   SRs = COMPACT Rmd (IS);;
int __builtin_compact_to_i4x8(long long int SRmd);
   BRs = COMPACT SRmd (I);;
int __builtin_compact_to_fr4x8(long long int SRmd);
   BRs = COMPACT SRmd;;
int __builtin_compact_to_fr4x8_trunc(long long int SRmd);
   BRs = COMPACT SRmd(T);;
int __builtin_compact_to_i4x8_sat(long long int SRmd);
   BRs = COMPACT SRmd(IS);
long long int __builtin_expand_fr2x16(int);
   Rsd = EXPAND SRm;;
long long int __builtin_expand_i2x16(int SRmd);
   Rsd = EXPAND SRm (I);;
long long int __builtin_expand_i4x8(int SRmd);
   SRsd = EXPAND BRm (I);;
```

Miscellaneous ALU Instructions:

```
long long int __builtin_llabs(long long int Rmd);
   LRsd = ABS Rmd;;
long long int __builtin_llavg(long long int Rmd, long long int Rnd);
```

```
   LRsd = (Rmd + Rnd) / 2;;

int __builtin_sum_2x16(int RM);
   Rs = SUM Rm;;
int __builtin_sum_2x32(long long int val);
   Rs = Rm + Rn;;
   where Rm is the low half of val and Rn is the high half of val
int __builtin_sum_4x16(long long int SRmd);
   Rs = SUM SRmd;;
int __builtin_sum_4x8(int BRm);
   Rs = SUM BRm;;
int __builtin_sum_8x8(long long int BRmd);
   Rs = SUM BRmd;;

long __builtin_lavg(long Rm, long Rn);
   Rs = (Rm + Rn) / 2;;
long __builtin_lavgt(long Rm, long Rn);
   Rs = (Rm + Rn) / 2 (T);;
long __builtin_lclip(long Rm, long Rn);
   Rs = CLIP Rm BY Rn;;
int __builtin_avg(int Rm, int Rn);
   Rs = (Rm + Rn) / 2;;
int __builtin_avgt(int Rm, int Rn);
   Rs = (Rm + Rn) / 2 (T);;

long long int __builtin_clip_4x16(long long int Rmd, long long int Rnd);
   SRsd = CLIP Rmd by Rnd;;
long long int __builtin_clip_8x8(long long int Rmd, long long int Rnd);
   BRsd = CLIP Rmd by Rnd;;
long long int __builtin_llclip(long long int Rmd, long long int Rnd);
   LRsd = CLIP Rmd by Rnd;;
int __builtin_clip(int Rm, int Rn);
   Rs = CLIP Rm by Rn;;
int __builtin_clip_2x16(int Rm, int Rn);
   SRs = CLIP Rm by Rn;;
int __builtin_clip_4x8(int Rm, int Rn);
   BRs = CLIP Rm by Rn;;

long long int __builtin_abs_4x16(long long int Rmd);
   SRsd = ABS Rmd;;
```

```
long long int __builtin_abs_8x8(long long int Rmd);
   BRsd = ABS Rmd;;
int __builtin_abs(int Rm);
   Rs = ABS Rm;;
int __builtin_abs_2x16(int Rm);
   SRs = ABS Rm;;
int __builtin_abs_4x8(int Rm);
   BRs = ABS Rm;;
int __builtin_neg_2x16_sat(int Rm);
   SRs = - Rm;;
int __builtin_neg_sat(int Rm);
   Rs = - Rm;;

long long int __builtin_neg_4x16_sat(long long int Rmd);
   SRsd = - Rmd;;
int __builtin_max(int Rm, int Rn);
   Rs = MAX (Rm, Rn);;
int __builtin_min(int Rm, int Rn);
   Rs = MIN (Rm, Rn);;
int __builtin_max_2x16(int Rm, int Rn);
   SRs = MAX (Rm, Rn);;
int __builtin_min_2x16(int Rm, int Rn);
   SRs = MIN (Rm, Rn;);
int __builtin_max_4x8(int Rm, int Rn);
   BRs = MAX (Rm, Rn);;
int __builtin_min_4x8(int Rm, int Rn);
   BRs = MIN (Rm, Rn);;

long long int __builtin_max_4x16(long long int Rmd, long long int Rnd);
   SRsd = MAX (Rmd, Rnd);;
long long int __builtin_min_4x16(long long int Rmd, long long int Rnd);
   SRsd = MIN (Rmd, Rnd);;
long long int __builtin_min_8x8(long long int Rmd, long long int Rnd);
   BRsd = MIN (Rmd, Rnd);;
long long int __builtin_max_8x8(long long int Rmd, long long int Rnd);
   BRsd = MAX (Rmd, Rnd);;
long long int __builtin_llmax(long long int Rmd, long long int Rnd);
   LRsd = MAX (Rmd, Rnd);;
long long int __builtin_llmin(long long int Rmd, long long int Rnd);
   LRsd = MIN (Rmd, Rnd);;
```

```
unsigned int __builtin_max_u2x16(unsigned int Rm, unsigned int Rn);
   SRs = MAX (Rm, Rn) (U);;
unsigned int __builtin_min_u2x16(unsigned int Rm, unsigned int Rn);
   SRs = MIN (Rm, Rn) (U);;
unsigned long long int __builtin_max_u4x16(unsigned long long int Rmd,
                                           unsigned long long int Rnd);
   SRsd = MAX (Rmd, Rnd) (U);;
unsigned long long int __builtin_min_u4x16(unsigned long long int Rmd,
                                           unsigned long long int Rnd);
   SRsd = MIN (Rmd, Rnd) (U);;

float __builtin_recip (float Rm);;
   Rs = RECIPS Rm;;
float __builtin_rsqrt (float Rm);;
   Rs = RSQRTS Rm;;
```

Shifter Instructions:

```
int __builtin_ashift_4x8(int Rm, int Rn);
   BRs = ASHIFT Rm BY Rn;    or BRs = ASHIFT Rm BY imm4;;
int __builtin_lshift_4x8(int Rm, int Rn);
   BRs = LSHIFT Rm BY Rn;    or BRs = LSHIFT Rm BY imm4;;
int __builtin_ashift_2x16(int Rm, int Rn);
   SRs = ASHIFT Rm BY Rn;    or SRs = ASHIFT Rm BY imm5;;
int __builtin_lshift_2x16(int Rm, int Rn);
   SRs = LSHIFT Rm BY Rn;    or SRs = LSHIFT Rm BY imm5;;
long long int __builtin_ashift_8x8(long long int Rmd, int Rnd);
   BRsd = ASHIFT Rmd BY Rnd; or BRs = ASHIFT Rmd BY imm4;;
long long int __builtin_lshift_8x8(long long int Rmd, int Rnd);
   BRsd = LSHIFT Rmd BY Rnd; or BRs = LSHIFT Rmd BY imm4;;
long long int __builtin_ashift_4x16(long long int Rmd, int Rnd);
   SRsd = ASHIFT Rmd BY Rnd; or SRs = ASHIFT Rmd BY imm5;;
long long int __builtin_lshift_4x16(long long int Rmd, int Rnd);
   SRsd = LSHIFT Rmd BY Rnd; or SRs = LSHIFT Rmd BY imm5;;
long long int __builtin_ashift_2x32(long long int Rmd, int Rnd);
   Rsd = ASHIFT Rmd BY Rnd;  or Rs = ASHIFT Rmd BY imm6;;
long long int __builtin_lshift_2x32(long long int Rmd, int Rnd);
   Rsd = LSHIFT Rmd BY Rnd;  or Rs = LSHIFT Rmd BY imm6;;
int __builtin_rotate_1x32(int Rm, int Rn);
   Rs = ROT Rm BY Rn;        or Rs = ROT Rm BY imm6;;
```

```
long long __builtin_rotate_2x32(long long int Rmd, int Rnd);
   Rsd = ROT Rmd BY Rnd;     or Rsd = ROT Rmd BY imm6;;
long long __builtin_rotate_1x64(long long int Rmd, int Rnd);
   LRsd = ROT Rmd BY Rnd;    or LRsd = ROT Rmd BY imm6;;
```

Bit Manipulation Instructions:

```
int __builtin_count_ones(int Rm);
   Rs = ONES Rm;;
int __builtin_lcount_ones(long Rm);
   Rs = ONES Rm;;
int __builtin_llcount_ones(long long int Rmd);
   Rs = ONES Rmd;;
int __builtin_lead_ones(int Rm);
   Rs = LD1 Rm;;
int __builtin_lllead_ones(long long int Rmd);
   Rs = LD1 Rm;;
int __builtin_lead_zero(int Rm);
   Rs = LD0 Rmd;;
int __builtin_lllead_zero(long long int Rmd);
   Rs = LD0 Rmd;;
long long int __builtin_merge_2x16(int Rm, int Rn);
   SRsd = MERGE Rm, Rn;;
long long int __builtin_merge_4x8(int Rm, int Rn);
   BRsd = MERGE Rm, Rn;;
int __builtin_fdep(int Rs, int Rn, int Rm);
   Rs += FDEP Rn BY Rm;;
int __builtin_fdep_se(int Rs, int Rn, int Rm);
   Rs += FDEP Rn BY Rm (SE);;
int __builtin_fdep_zf(int Rs, int Rn, int Rm);
   Rs += FDEP Rn BY Rm (ZF);;
int __builtin_fdep_long_control(int Rs, int Rn, long long int Rmd);
   Rs += FDEP Rn BY Rmd;;
int __builtin_fdep_long_control_se(int Rs, int Rn, long long int Rmd);
   Rs += FDEP Rn BY Rmd (SE);;
int __builtin_fdep_long_control_zf(int Rs, int Rn, long long int Rmd);
   Rs += FDEP Rn BY Rmd (ZF);;
long long int __builtin_fdep2(long long int Rsd,
                              long long int Rnd, int Rm);
   LRsd += FDEP Rnd BY Rm;;
long long int __builtin_fdep2_se(long long int Rsd,
```

```
                                    long long int Rnd, int Rm);
   LRsd += FDEP Rnd BY Rm (SE);;
long long int __builtin_fdep2_zf(long long int Rsd,
                                    long long int Rnd, int Rm);
   LRsd += FDEP Rnd BY Rm (ZF);;
long long int __builtin_fdep2_long_control(long long int Rsd,
                                           long long int Rnd,
                                           long long int Rmd);
   LRsd += FDEP Rnd BY Rmd;;
long long int __builtin_fdep2_long_control_se(long long int Rsd,
                                           long long int Rnd,
                                           long long int Rmd);
   Lsd += FDEP Rnd BY Rmd (SE);;
long long int __builtin_fdep2_long_control_zf(long long int Rsd,
                                           long long int Rnd,
                                           long long int Rmd);
   LRsd += FDEP Rnd BY Rmd (ZF);;
int __builtin_fext(int Rm, int Rn);
   Rs = FEXT Rm by Rn (SE);;
int __builtin_fext_se(int Rm, int Rn);
   Rs = FEXT Rm by Rn (SE);;
int __builtin_fext_ze(int Rm, int Rn);
   Rs = FEXT Rm by Rn;;
int __builtin_fext_long_control(int Rm, long long int Rnd);
   Rs = FEXT Rm by Rnd (SE);;
int __builtin_fext_long_control_se(int Rm, long long int Rnd);
   Rs = FEXT Rm by Rnd (SE;;
int __builtin_fext_long_control_ze(int Rm, long long int Rnd);
   Rs = FEXT Rm by Rnd;;
long long int __builtin_fext2(long long int Rmd, int Rn);
   LRsd = FEXT Rmd by Rn (SE);;
long long int __builtin_fext2_se(long long int Rmd, int Rn);
   LRsd = FEXT Rmd by Rn (SE);;
long long int __builtin_fext2_ze(long long int Rmd, int Rn);
   LRsd = FEXT Rmd by Rn;;
long long int __builtin_fext2_long_control(long long int Rmd,
                                          long long int Rnd);
   LRsd = FEXT Rmd by Rnd (SE);;
long long int __builtin_fext2_long_control_se(long long int Rmd,
                                             long long int Rnd);
```

```
   LRsd = FEXT Rmd by Rnd (SE);;
long long int __builtin_fext2_long_control_ze(long long int Rmd,
                                              long long int Rnd);
   LRsd = FEXT Rmd by Rnd;;
int __builtin_exp(int Rn);
   Rs = EXP Rn;;
int __builtin_exp2(long long int Rnd);
   Rs = EXP Rnd;;
```

Multiplier Instructions:

```
unsigned int __builtin_mult_u2x16(unsigned int Rmd, unsigned int Rnd);
   Rsd = Rmd * Rnd (IU);;
long long int __builtin_mult_i2x16_wide(int Rmd, int Rnd);
   Rsq = Rmd * Rnd (I);;
unsigned long long int __builtin_mult_u2x16_wide(
                                    long long int Rmd,
                                    long long int Rnd);
   Rsq = Rmd * Rnd (IU);;
long long int __builtin_mult_i4x16(long long int Rmd,
                                   long long int Rnd);
   Rsd = Rmd * Rnd (I);;
long long int __builtin_mult_i4x16_sat(long long int Rmd,
                                       long long int Rnd);
   Rsd = Rmd * Rnd (IS);;
unsigned long long int __builtin_mult_u4x16(unsigned long long int RMD,
                                       unsigned long long int Rnd);
   Rsd = Rmd * Rnd (IU);;
__builtin_quad __builtin_mult_i4x16_wide(long long int Rmd,
                                    long long int Rnd);
   Rsq = Rmd * Rnd (I);;
__builtin_quad __builtin_mult_u4x16_wide(unsigned long long int Rmd,
                                       unsigned long long int Rnd);
   Rsq = Rmd * Rnd (IU);;
int __builtin_multr_fr1x32(int Rm_1, int Rm_2);
   Rs = Rm_1 * Rm_2;;
int __builtin_multr_fr1x32_sat(int Rm_1, int Rm_2);
   Rs = Rm_1 * Rm_2 (S);;
int __builtin_mult_fr1x32(int Rm_1, int Rm_2);
   Rs = Rm_1 * Rm_2 (T);;
int __builtin_mult_fr1x32_sat(int Rm_1, int Rm_2);
```

```
   Rs = Rm_1 * Rm_2 (TS);;
long long int __builtin_mult_i1x32_wide(int Rm, int Rn);
   Rsd = Rm * Rn (I);
unsigned long long int __builtin_mult_u1x32_wide(unsigned int Rm,
unsigned int Rn);
   Rsd = Rm * Rn (UI);;
int __builtin_mult_i2x16(int Rmd, int Rnd);
   Rsd = Rmd * Rnd (I);;
int  __builtin_mult_i2x16_sat(int Rm,int Rn);
   Rsd = Rmd * Rnd (IS);;
int __builtin_mult_sat(int Rm, int Rn);
   Rs = Rm * Rn (IS);;
int __builtin_multr_fr2x16(int Rmd, int Rnd);
   Rsd = Rmd * Rnd;;
int __builtin_multr_fr2x16_sat(int Rmd, int Rnd);
   Rsd = Rmd * Rnd (S);;
int __builtin_mult_fr2x16(int Rmd, int Rnd);
   Rsd = Rmd * Rnd (T);;
int __builtin_mult_fr2x16_sat(int Rmd, int Rnd);
   Rsd = Rmd * Rnd (TS);;
long long int __builtin_mult_fr4x16(long long int Rmd,
                                    long long int Rnd);
   Rsd = Rmd * Rnd (T);;
long long int __builtin_mult_fr4x16_sat(long long int Rmd,
                                        long long int Rnd);
   Rsd = Rmd * Rnd (ST);;
long long int __builtin_multr_fr4x16(long long int Rmd,
                                     long long int Rnd);
   Rsd = Rmd * Rnd;;
long long int __builtin_multr_fr4x16_sat(long long int Rmd,
                                         long long int Rnd);
   Rsd = Rmd * Rnd (S);;
int __builtin_cmult_i2x16(int Rm, int Rn);
   MRa += Rm ** Rn (CI);;
   SRsd = COMPACT MR3:0 (I);;
int __builtin_cmult_i2x16_sat(int Rm, int Rn);
   MRa += Rm ** Rn (CI);;
   SRsd = COMPACT MR3:0 (IS);;
long long int __builtin_cmult_i2x16_wide(int Rm, int Rn);
   MRa += Rm ** Rn (CI);;
```

```
   Rsd = MRa;;
int __builtin_cmult_conj_i2x16(int Rm, int Rn);
   MRa += Rm ** Rn (CIJ);;
   SRsd = COMPACT MR3:0 (I);;
int __builtin_cmult_conj_i2x16_sat(int Rm, int Rn);
   MRa += Rm ** Rn (CIJ);;
   SRsd = COMPACT MR3:0 (IS);;
long long int __builtin_cmult_conj_i2x16_wide(int Rm, int Rn);
   MRa += Rm ** Rn (CIJ);;
   Rsd = MRa;;
int __builtin_cmult_fr2x16(int Rm, int Rn);
   MRa += Rm ** Rn (C);;
   SRsd = COMPACT MR3:0;;
int __builtin_cmult_conj_fr2x16(int Rm, int Rn);
   MRa += Rm ** Rn (CJ);;
   SRsd = COMPACT MR3:0;;
int __builtin_cmult_conj_fr2x16_sat(int Rm, int Rn);
   MRa += Rm ** Rn (CJ);;
   SRsd = COMPACT MR3:0 (S);;
int __builtin_cmult_fr2x16_sat(int Rm, int Rn);
   MRa += Rm ** Rn (C);;
   SRsd = COMPACT MR3:0 (S);;
int __builtin_cmultr_fr2x16(int Rm, int Rn);;
   MRa += Rm ** Rn (CR);;
   SRsd = COMPACT MR3:0;;
int __builtin_cmultr_fr2x16_sat(int Rm, int Rn);;
   MRa += Rm ** Rn (CR);;
   SRsd = COMPACT MR3:0 (S);;
int __builtin_cmultr_conj_fr2x16(int Rm, int Rn);;
   MRa += Rm ** Rn (CRJ);;
   SRsd = COMPACT MR3:0;;
int __builtin_cmultr_conj_fr2x16_sat(int Rm, int Rn);;
   MRa += Rm ** Rn (CRJ);;
   SRsd = COMPACT MR3:0 (S);;
```

Floating-Point Operations:

```
int __builtin_conv_fix(float FRm);
   Rs = FIX FRm;;
int __builtin_conv_FtoR(float FRm);
```

```
   Rs = FIX FRm by 31;;
float __builtin_conv_RtoF(int Rm);
   FRs = FLOAT Rm by -31;;
float __builtin_copysignf(float Rm, float Rn);
   FRs = Rm COPYSIGN Rn;;
float __builtin_fabsf(float Rm);
   FRs = ABS Rm;;
float __builtin_favgf(float Rm, float Rn);
   FRs = (Rm + Rn) / 2;;
float __builtin_fclipf(float Rm, float Rn);
   FRs = CLIP Rm by Rn;;
float __builtin_fmaxf(float Rm, float Rn);
   FRs = MAX (Rm, Rn);;
float __builtin_fminf(float Rm, float Rn);
   FRs = MIN (Rm, Rn;;
float __builtin_recip (float Rm);
   FRs = RECIPS Rm;;
float __builtin_rsqrt (float Rm);
   FRs = RSQRTS Rm;;
```

Miscellaneous:

```
void __builtin_idle(void);
   IDLE;;
void __builtin_idle_lp(void);
   IDLE (LP);;
void __builtin_assert(int);    ignored
```

Memory Allocation:

```
void *__builtin_alloca_aligned(int size, int align);
```

Allocate "`size`" words of aligned data on the stack with alignment "`align`".

Composition and Decomposition:

The following functions are used to compact or merge values to create larger or smaller types. In some cases, the optimizer is able remove instructions that may be generated by these functions.

To combine smaller objects into a single larger object, use

```
__builtin_quad compose_128(long long ,long long);
      Compose a 128-bit datum
long long int __builtin_compose_64(int hi, int lo);
      Compose a 64-bit datum
unsigned long long compose_64u(unsigned int ,unsigned int);
      Compose an unsigned 64-bit datum
long double f_compose_64(float ,float);
      Compose a double precision datum from two single-precision
```

To create an object of size `n` by `s` (`n` number of parts of size `s` bits), use

```
int compact_to_fr2x16(long long);
      Create a 2x16 fractional item from a 2x32 fractional item
int compact_to_i4x8(long long);
      Create a 4x8 integer item from a 4x16 integer item
```

To expand an object of size `n` by `s` into the next size up, use

Into two `fracts`:

```
long long expand_fr2x16(int );
```

Into two 2x16s:

```
long long expand_i4x8(int );
```

To extract the low/high half-out of various larger objects, use

```
int __builtin_low_32(long long int);
      extract least significant word
unsigned int __builtin_low_32u(unsigned long long int);
      extract least significant word
int __builtin_high_32(long long int);
      extract most significant word
unsigned int __builtin_high_32u(unsigned long long int);
      extract most significant word
long long int __builtin_low_64(__builtin_quad);
      extract least significant words
long long int __builtin_high_64(__builtin_quad);
      extract most significant words
```

```
float __builtin_f_low_32(long double);
      extract least significant word from a 64-bit float
float __builtin_f_high_32(long double);
      extract most significant word from a 64-bit float
```

System Register Access:

To access to various system registers, use

```
int __builtin_sysreg_read(int reg);
      read word-sized system register
long long int __builtin_sysreg_read2(int reg);
      read double-word-sized system register
__builtin_quad __builtin_sysreg_read4(int reg);
      read quad-word-sized system register
void __builtin_sysreg_write(int value, unsigned int value);
      write to word-sized system register
void __builtin_sysreg_write2(int, unsigned long long int value);
      write to double-word-sized system register
void __builtin_sysreg_write4(int, __builtin_quad value);
      write to quad-word-sized system register
```

The register names are defined in `sysreg.h` and must be a literal used at compile time. The register numbers defined in `sysreg.h` are the compiler's internal register numbers. The effect of using the incorrect function for the size of the register or using a bad register number is undefined.

Use of the `__XSTAT` and `__YSTAT` registers in the `sysreg` built-in functions are deprecated. There is no mechanism by which the source can associate a read of the `STAT` registers with a specific operation that may update the `STAT` registers. An inline asm can be used to get access to these registers if needed: the read of a `STAT` register can explicitly be placed where it is required.

The `__read_ccnt` macro reads the value in each of the cycle count registers and uses them to create a single 64-bit integer value. The macro is defined in `sysreg.h` as:

```
#define __read_ccnt()
(__builtin_compose_64(__builtin_sysreg_read(__UNDER__(CCNT0)),
__builtin_sysreg_read(__UNDER__(CCNT1))))
```

Data Alignment Buffer (DAB) Built-in Functions

The DAB built-in functions provide access to the Data Alignment Buffer functionality. Two sets of built-in functions are provided. The first covers 32-bit data items and the built-ins are the same whether byte-addressing mode is enabled (with the "-char-size-{8|32}" switch) or not. The second set covers 16-bit data items and the details of the use depend on whether byte-addressing mode is enabled or not.

The following DAB built-in functions are provided.

```
   long long __builtin_dab_2x32(int **a);
   __builtin_quad __builtin_dab_4x32(int **a);
#ifdef __TS_BYTE_ADDRESS
   // length given in terms of 16-bit items
   int __builtin_dab_2x16(short **a);
   long long __builtin_dab_4x16(short **a);
   __builtin_quad __builtin_dab_8x16(short **a);
#else

   // length given in terms of 16-bit items
   int __builtin_dab_2x16(int **a, int offset);
   long long __builtin_dab_4x16(int **a, int offset);
   __builtin_quad __builtin_dab_8x16(int **a, int offset);
#endif
```

Every DAB built-in function takes a pointer to a pointer to the data as the first argument. The data pointer can in this way be automatically post-incremented by the execution of the built-in function according to the type and number of data items read. For the 16-bit DAB intrinsics, in byte-addressing mode the data pointer is a `short` integer pointer, and can thus address a given 16-bit quantity. In word-addressing mode, pointers

do not have the resolution to address a part-word entity, so a second argument needs to be supplied which gives the half-word offset from the word-aligned address.

Here is an example of the 32-bit DAB built-in functions:

```
void func1 (__builtin_quad *a, int *b, int n) {
   int i;
   for (i=0; i<n; i++) {
      a[i] = __builtin_dab_4x32 (&b);
   }
}
```

Here is an example of the 16-bit DAB built-in functions for use in byte-addressing mode:

```
void func2 (long long int *a, short int *b, int n) {
   int i;
   for (i=0; i<n; i++) {
      a[i] = __builtin_dab_4x16 (&b);
   }
}
```

Here is an example of the 16-bit DAB built-in functions for use in word-addressing mode:

```
void func3 (long long int *a, int *b, int offset, int n) {
   int i;
   for (i=0; i<n; i++) {
      a[i] = __builtin_dab_4x16 (&b, offset);
   }
}
```

Circular Buffer Data Alignment Buffer (DAB) Built-in Functions

The circular buffer DAB intrinsics are similar to the DAB intrinsics (described in the "Data Alignment Buffer (DAB) Built-in Functions" on page 1-161), only with extra "*base*" and "*length*" parameters. The automatic post-increments now become post-increment with wrap-around according to the circular buffering.

Therefore,

```
   long long __builtin_dabcb_2x32(int **a, int *base,
                                  int length);
   __builtin_quad __builtin_dabcb_4x32(int **a,
                                       int *base, int length);
#ifdef __TS_BYTE_ADDRESS

// length given in terms of 16-bit items
   int __builtin_dabcb_2x16(short **a, short *base, int length);
   long long __builtin_dabcb_4x16(short **a, short *base,
                                  int length);
  __builtin_quad __builtin_dabcb_8x16(short **a, short *base,
                                      int length);
#else

// length given in terms of 16-bit items
   int __builtin_dabcb_2x16(int **a, int offset, int *base,
                            int length);
   long long __builtin_dabcb_4x16(int **a, int offset,
                                  int *base, int length);
   __builtin_quad __builtin_dabcb_8x16(int **a, int offset,
                                       int *base, int length);
#endif
```

where "`a`" is the address of the pointer to be loaded from.

It should be noted that circular buffer with the DAB or SDAB only works when

- "`base`" is a quad-word aligned address, and
- the `length` is a multiple of quad-words.

The compiler may perform some checks on this where possible but in the case of non-constants cannot be expected to verify these conditions.

The compiler assumes that the initial value of `*a` or `*a + offset` is between `base` (inclusive) and `base+length` (not inclusive). In the case of the word-addressed compiler, circular buffering ensures that the value of `*a + offset` is always in this range (and not necessarily `*a` itself).

Unoptimized, the generated code performs two DAB accesses (the prime and the read) as well as initialization and reset of the length and base registers for each DAB intrinsic. When optimization is turned on, unnecessary setting of length and base registers and the DAB prime operation is removed or hoisted out of loops to generate the optimal code.

Communications Logic Unit Operations

The following is a description of the built-in functions used to generate instructions to be executed by the Communications Logic Unit (CLU). They are grouped for clarity into `MAX / TMAX`, `PERMUTE`, `ACS`, `DESPREAD`, `RECIPS`, `RSQRTS`, and `XCORRS`.

For instructions that have more than one output, there is a built-in function for each of these outputs. In such cases, the built-in functions for the second or third results need to be chained through the result of the first built-in function. For example, in the case of `DESPREAD`:

```
r = __builtin_despread (q, c, d);
th = __builtin_despread_res2 (r);
```

This is somewhat cumbersome and the compiler is unforgiving if errors are introduced (such as forgetting the second built-in function), so the `builtins.h` file includes macros for each of the instructions that return multiple results. In the case of the example above, use the following form:

```
__despread (q, c, d, r, th);
```

The variables r and th are outputs even though they appear as arguments to `__despread`. Looking at the definition of `__despread` from `builtins.h` makes this more obvious:

```
#define __despread(I1,I2,I3,O1,O2) {       \
   O1 = __builtin_despread(I1,I2,I3);      \
   O2 = __builtin_despread_res2(O1);       \
}
```

The following examples for the instructions that have multiple results use the shorthand form, and the inputs and outputs are described.

TMAX, TMAX_ADD, TMAX_SUB, MAX_ADD, MAX_SUB

This section also covers the (S) variants of these instructions which deal with 16-bit components.

Instruction:

```
Rs = TMAX (TRm, TRn)
```

Builtin:

```
int __builtin_tmax (int, int);
```

Example:

```
int r, a, b;
r = __builtin_tmax (a, b);
```

Description:

`r` corresponds to `Rs`

`a` corresponds to `TRm`

`b` corresponds to `TRn`

Instruction:

```
SRs = TMAX (TRm, TRn)
```

Builtin:

```
int __builtin_tmax_4s (int, int);
```

Example:

```
int r, a, b;
r = __builtin_tmax_4s (a, b);
```

Description:

`r` corresponds to `Rs`

`a` corresponds to `TRm`

`b` corresponds to `TRn`

Instruction:

```
TRsd = TMAX (TRmd + Rmq_h, TRnd + Rmq_l)
```

Builtin:

```
long long int __builtin_tmax_add (long long int,
                                  long long int,
                                  __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_tmax_add (a, b, q);
```

Description:

`r` corresponds to TRsd

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

Instruction:

```
TRsd = TMAX (TRmd - Rmq_h, TRnd - Rmq_l)
```

Builtin:

```
long long int __builtin_tmax_sub (long long int,
                                  long long int,
                                  __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_tmax_sub (a, b, q);
```

Description:

`r` corresponds to `TRsd`

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

Instruction:

```
TRsd = MAX (TRmd + Rmq_h, TRnd + Rmq_l)
```

Builtin:

```
long long int __builtin_max_add (long long int,
                                 long long int,
                                 __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_max_add (a, b, q);
```

Description:

`r` corresponds to `TRsd`

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

Instruction:

```
TRsd = MAX (TRmd - Rmq_h, TRnd - Rmq_l)
```

Builtin:

```
long long int __builtin_max_sub (long long int,
                                 long long int,
                                 __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_max_sub (a, b, q);
```

Description:

`r` corresponds to `TRsd`

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

Instruction:

```
STRsd = TMAX (TRmd + Rmq_h, TRnd + Rmq_l)
```

Builtin:

```
long long int __builtin_tmax_add_8s (long long int,
                                     long long int,
                                     __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_tmax_add_8s (a, b, q);
```

Description:

`r` corresponds to `TRsd`

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

Instruction:

```
STRsd = TMAX (TRmd - Rmq_h, TRnd - Rmq_l)
```

Builtin:

```
long long int __builtin_tmax_sub_8s (long long int,
                                     long long int,
                                     __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_tmax_sub_8s (a, b, q);
```

Description:

`r` corresponds to `TRsd`

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

Instruction:

```
STRsd = MAX (TRmd + Rmq_h, TRnd + Rmq_l)
```

Builtin:

```
long long int __builtin_max_add_8s (long long int,
                                    long long int,
                                    __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_max_add_8s (a, b, q);
```

Description:

`r` corresponds to `TRsd`

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

Instruction:

```
STRsd = MAX (TRmd - Rmq_h, TRnd - Rmq_l)
```

Builtin:

```
long long int __builtin_max_sub_8s (long long int,
                                    long long int,
                                    __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_max_sub_8s (a, b, q);
```

Description:

`r` corresponds to `TRsd`

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`q` corresponds to `Rmq`

PERMUTE

Instruction:

```
Rsd = PERMUTE (Rmd, Rn)
```

Builtin:

```
long long int __builtin_permute_8b (long long int, int);
```

Example:

```
long long int r, a;
int b;
r = __builtin_permute_8b (a, b);
```

Description:

`r` corresponds to `Rsd`

`a` corresponds to `Rmd`

`b` corresponds to `Rnd`

Instruction:

```
Rsq = PERMUTE (Rmd, -Rmd, Rn)
```

Builtin:

```
__builtin_quad __builtin_permute_8s (long long int, int);
```

Example:

```
__builtin_quad r;
long long int a;
int b;
r = __builtin_permute_8s (a, b);
```

Description:

`r` corresponds to `Rsd`

`a` corresponds to `Rmd`

`b` corresponds to `Rnd`

ACS

Instruction:

```
TRsq = ACS (TRmd, TRnd, Rm) (TMAX)
```

Builtins:

```
__builtin_quad __builtin_acs_tmax (long long int,
                                   long long int,
                                   int,
                                   long long int);
long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_tmax (long long int,
            long long int,
            int,
            long long int,
            __builtin_quad,
            long long int);
```

Example:

```
__builtin_quad r;
long long int a, b, thi, tho;
int c;
__acs_tmax (a, b, c, thi, r, tho);
```

Description:

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`c` corresponds to `Rm`

`thi` corresponds to Trellis History Registers before execution (input)

`r` corresponds to `TRsq`

`tho` corresponds to Trellis History Registers after execution (output)

Instruction:

```
TRsq = ACS (TRmd, TRnd, Rm)
```

Builtins:

```
__builtin_quad __builtin_acs_max (long long int,
                                  long long int,
                                  int,
                                  long long int);
long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_max (long long int,
           long long int,
           int,
           long long int,
           __builtin_quad,
           long long int);
```

Example:

```
__builtin_quad r;
long long int a, b, thi, tho;
int c;
__acs_max (a, b, c, thi, r, tho);
```

Description:

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`c` corresponds to `Rm`

`thi` corresponds to Trellis History Registers before execution (input)

`r` corresponds to `TRsq`

`tho` corresponds to Trellis History Registers after execution (output)

Instruction:

```
STRsq = ACS (TRmd, TRnd, Rm) (TMAX)
```

Builtins:

```
__builtin_quad __builtin_acs_tmax_8s (long long int,
                                      long long int,
                                      int,
```

```
                                          long long int);
long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_tmax_8s (long long int,
               long long int,
               int,
               long long int,
               __builtin_quad,
               long long int);
```

Example:

```
__builtin_quad r;
long long int a, b, thi, tho;
int c;
__acs_tmax_8s (a, b, c, thi, r, tho);
```

Description:

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`c` corresponds to `Rm`

`thi` corresponds to Trellis History Registers before execution (input)

`r` corresponds to `TRsq`

`tho` corresponds to Trellis History Registers after execution (output)

Instruction:

```
STRsq = ACS (TRmd, TRnd, Rm)
```

Builtins:

```
__builtin_quad __builtin_acs_max_8s (long long int,
                                     long long int,
                                     int,
                                     long long int);
long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_max_8s (long long int,
              long long int,
              int,
              long long int,
              __builtin_quad,
              long long int);
```

Example:

```
__builtin_quad r;
long long int a, b, thi, tho;
int c;
__acs_max_8s (a, b, c, thi, r, tho);
```

Description:

`a` corresponds to `TRmd`

`b` corresponds to `TRnd`

`c` corresponds to `Rm`

`thi` corresponds to Trellis History Registers before execution (input)

`r` corresponds to `TRsq`

`tho` corresponds to Trellis History Registers after execution (output)

DESPREAD

Instruction (ADSP-TS101 processor):

```
TRs = DESPREAD (Rmq, THrd) + TRn
```

Instruction (ADSP-TS201 processor):

```
TRs += DESPREAD (Rmq, THrd)
```

Builtins:

```
int __builtin_despread (__builtin_quad, long long int, int);
long long int __builtin_despread_res2 (int);
```

Shorthand:

```
__despread (__builtin_quad,
            long long int,
```

```
                int,int,
                long long int);
```

Example:

```
__builtin_quad q;
long long int thi, tho;
int d;
int r;
__despread (q, thi, d, r, tho);
```

Description:

`q` corresponds to `Rmq`

`thi` corresponds to Trellis History Registers before execution (input)

`d` corresponds to `TRn`

`r` corresponds to `TRs`

`tho` corresponds to Trellis History Registers after execution (output)

Instruction (ADSP-TS101 processor):

```
TRs = DESPREAD (Rmq, THrd) + TRn
```

Instruction (ADSP-TS201 processor):

```
TRs += DESPREAD (Rmq, THrd)
```

Builtins:

```
int __builtin_despread_i (__builtin_quad, long long int, int);
long long int __builtin_despread_res2 (int);
```

Shorthand:

```
__despread_i (__builtin_quad,
              long long int,
              int,int,
              long long int);
```

Example:

```
__builtin_quad q;
long long int thi, tho;
int d;
```

```
int r;
__despread (q, thi, d, r, tho);
```

Description:

In this version, `THrd` is loaded using the (`i`) option to interleave the input bits. Obviously, when loading the spreading codes interleaved into the Trellis History registers for the first `DESPREAD`, the `__despread_i` built-in function should be used. When chaining the shifted spreading codes from the first `DESPREAD` into a subsequent `DESPREAD`, use the `__despread` built-in function, otherwise the Trellis History registers after the first `DESPREAD` is stored and then reloaded (with interleave) again.

`q` corresponds to `Rmq`

`thi` corresponds to Trellis History Registers before execution (input)

`d` corresponds to `TRn`

`r` corresponds to `TRs`

`tho` corresponds to Trellis History Registers after execution (output)

XCORRS

Instruction (ADSP-TS201 processor):

```
TR15:0/31:16 = XCORRS (Rmq, THRq) (CUT #imm) (CLR) (EXT)
```

Builtins:

```
__builtin_quad __builtin_xcorrs (__builtin_quad signal,
                                 long long int codeslo,
                                 long long int codeshi,
                                 int cut,
                                 __builtin_quad accum0,
                                 __builtin_quad accum1,
                                 __builtin_quad accum2,
                                 __builtin_quad accum3);
__builtin_quad __builtin_xcorrs_res2 (__builtin_quad);
__builtin_quad __builtin_xcorrs_res3 (__builtin_quad);
__builtin_quad __builtin_xcorrs_res4 (__builtin_quad);
long long int __builtin_xcorrs_res5 (__builtin_quad);
long long int __builtin_xcorrs_res6 (__builtin_quad);
```

Shorthand:

```
void __xcorrs (__builtin_quad signal,
               long long int *codeslo,
               long long int *codeshi,
               int cut,
               int *accum);
```

Example:

```
__builtin_quad signal;
long long int codeslo;
long long int codeshi;
__builtin_quad accums[4];
__xcorrs (signal, &codeslo, &codeshi, /* cut */ 0, &accums);
```

Description:

`signal` corresponds to `Rmq`

`codeslo` and `codeshi` correspond to low and high halves of `THRq` (input and output)

`accums` corresponds to `TR15:0` or `TR31:16` depending on which half of the register file is being used.

The contents of `accums` are copied into the Trellis registers before the instruction, and the accumulations copied back into accums after execution. The compiler optimizes sequences of `XCORRS` to remove unnecessary transfers in and out of the Trellis registers, as it does for the other enhanced communication built-in functions.

The same thing happens for the Trellis History registers, which are loaded from the two double words pointed to by `codeslo` and `codeshi`, and stored back into these locations after execution.

The `accums` pointer must be quad aligned, and the `codeslo` and `codeshi` pointers must be double-word aligned. These alignments should be declared with `__builtin_aligned` where necessary. If this is not done, the behavior of the compiler, and the resulting code, is undefined.

The `cut` argument must be an immediate value.

The following shorthand forms are for variants of the XCORRS instruction which specify the (EXT) and/or (CLR) options.

```
void __xcorrs_clr (__builtin_quad signal,
                   long long int *codeslo,
                   long long int *codeshi,
                   int cut,
                   int *accum);
void __xcorrs_ext (__builtin_quad signal,
                   long long int *codeslo,
                   long long int *codeshi,
                   int cut,
                   int *accum);
void __xcorrs_clr_ext (__builtin_quad signal,
                       long long int *codeslo,
                       long long int *codeshi,
                       int cut,
                       int *accum);
```

At a low level, these macros make use of alternative built-in functions for the primary result and the same built-in functions as above for the subsequent results.

There are also _i versions of the built-in function:

```
void __xcorrs_i (__builtin_quad signal,
                 long long int *codeslo,
                 long long int *codeshi,
                 int cut,
                 int *accum);
void __xcorrs_i_clr (__builtin_quad signal,
                     long long int *codeslo,
                     long long int *codeshi,
                     int cut,
                     int *accum);
void __xcorrs_i_ext (__builtin_quad signal,
                     long long int *codeslo,
                     long long int *codeshi,
                     int cut,
                     int *accum);
```

```
void __xcorrs_i_clr_ext (__builtin_quad signal,
                         long long int *codeslo,
                         long long int *codeshi,
                         int cut,
                         int *accum);
```

For the _i versions, the codeshi value is loaded into the upper Trellis History registers with the (i) option, to interleave the input bits of the spreading code.

Pragmas

The compiler supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. There are two types of pragma usage: pragma directives and pragma operators.

Pragma directives have the following syntax:

```
#pragma pragma-directive pragma-directive-operands new-line
```

Pragma operators have the following syntax:

```
_Pragma ( string-literal )
```

When processing a pragma operator, the compiler effectively turns it into a pragma directive using a non-string version of *string-literal*. This means that the following pragma directive

```
#pragma linkage_name mylinkname
```

can also be equivalently be expressed using the following pragma operator

```
_Pragma ("linkage_name mylinkname")
```

The examples in this manual use the directive form.

The C/C++ compiler issues a warning when it encounters an unrecognized pragma directive or pragma operator. The C/C++ compiler does not expand any pre-processor macros used within any pragma directive or pragma operator.

The C compiler supports pragmas for:

- Arranging alignment of data
- Defining functions that can act as interrupt handlers
- Changing the optimization level, midway through a module
- Changing how an externally visible function is linked
- Header file configurations and properties
- Giving additional information about loop usage to improve optimizations

The following sections describe the pragmas that support the features listed above.

- "Data Alignment Pragmas" on page 1-181
- "Interrupt Handler Pragmas" on page 1-185
- "Loop Optimization Pragmas" on page 1-187
- "Function Side-Effect Pragmas" on page 1-193
- "General Optimization Pragmas" on page 1-203
- "Inline Control Pragmas" on page 1-204
- "Linking Control Pragmas" on page 1-206
- "Class Conversion Optimization Pragmas" on page 1-216
- "Template Instantiation Pragmas" on page 1-219
- "Header File Control Pragmas" on page 1-221
- "Diagnostic Control Pragmas" on page 1-223
- "Memory Bank Pragmas" on page 1-226

Refer to Chapter 2, "Achieving Optimal Performance from C/C++ Source Code", on how to use pragmas for code optimization.

Data Alignment Pragmas

The data alignment pragma is used to modify how the compiler arranges data within memory. Since the TigerSHARC processor architecture requires memory accesses to be naturally aligned, each data item is normally aligned at least as strongly as itself—double-word items have an alignment of 2 and quad-word items have and alignment of 4.

In byte-addressing mode, the same rule applies—two-byte `shorts` have an alignment of 2 (bytes) and four-byte `longs` have an alignment of 4 (bytes). When structs are defined, the struct's overall alignment is the same as the field which has the largest alignment. The struct's size may need padding to ensure all fields are properly aligned, and that the struct's overall size is a multiple of its alignment.

Sometimes, it is useful to change these alignments, for instance a `struct` may have its alignment increased to improve the compiler's opportunities in vectorizing access to the data.

Alignments specified for data alignment must be a power of 2.

Note that the minimum alignment allowed for structs in byte-addressing mode is 4 (32 bits).

#pragma align *num*

This pragma may be used before variable and field declarations. It applies to the variable or field declaration that immediately follows the pragma. The pragma's effect is that the next variable or field declaration is aligned on a boundary specified by *num*.

- If the pragma is being applied to a local variable then, since local variables are stored on the stack, the alignment of the variable will only be changed when *num* is not greater than the stack alignment (that is, 4 words). If *num* is greater than the stack alignment, then a warning is given that the pragma is being ignored.
- If *num* is greater than the alignment normally required by the following variable or field declaration, then the variable or field declaration's alignment is changed to *num*.
- If *num* is less than alignment normally required, then the variable or field declaration's alignment is changed to *num*, and a warning is given that the alignment has been reduced. For example,

```
typedef struct {
#pragma align 4
   int foo;
   int bar;
#pragma align 4
   int baz;
} aligned_ints;
```

In this example, any object declared of type `aligned_ints` has the `foo` and `baz` members aligned on quad-word boundaries. The size of this structure is eight words with two words unused between `bar` and `baz`, and three words of padding at the end of the structure. In byte addressing mode, the `NUM` argument to align would need to be 16 and not 4, as the argument to align is in addressable units.

The pragma also allows the following keywords as allowable alignment specifications:

_WORD – Specifies a 32-bit alignment
_LONG – Specifies a 64-bit alignment
_QUAD – Specifies a 128-bit alignment

These keywords specify the same alignment irrespective of the addressing mode specified.

The `align` pragma only applies to the immediately-following definition, even if that definition is part of a list. For example,

```
#pragma align 8
int i1, i2, i3;       // pragma only applies to i1
```

#pragma alignment_region (*alignopt*)

Sometimes it is desirable to specify an alignment for a number of consecutive data items rather than individually. This can be done using the `alignment_region` and `alignment_region_end` pragmas:

- `#pragma alignment_region` sets the alignment for all following data symbols up to the corresponding `alignment_region_end` pragma
- `#pragma alignment_region_end` removes the effect of the active alignment region and restores the default alignment rules for data symbols.

The rules concerning the argument are the same as for `#pragma align`. The compiler faults an invalid alignment (such as an alignment that is not a power of two). The compiler warns if the alignment of a data symbol within the control of an `alignment_region` is reduced below its natural alignment (as for `#pragma align`).

Use of the `align` pragma overrides the region alignment specified by the currently active `alignment_region` pragma (if there is one). The currently active `alignment_region` does not affect the alignment of fields.

Example:

```
#pragma align 4

int aa;          /* alignment 4 */
int bb;          /* alignment 1 */

#pragma alignment_region (2)

int cc;          /* alignment 2 */
int dd;          /* alignment 2 */
int ee;          /* alignment 2 */

#pragma align 4

int ff;          /* alignment 4 */
int gg;          /* alignment 2 */
int hh;          /* alignment 2 */

#pragma alignment_region_end

int ii;          /* alignment 1 */

#pragma alignment_region (1)

long double jj;   /* alignment 1, but the compiler warns
                     about the reduction */

#pragma alignment_region_end

#pragma alignment_region (3)
long double kk;   /* the compiler faults this, alignment is not
                     a power of two */

#pragma alignment_region_end
```

Interrupt Handler Pragmas

The interrupt pragmas provide a method by which the user can write interrupt service routines in C and install them directly into the interrupt vector table, bypassing the dispatcher provided with the C run time library.

Marking a routine with an `interrupt` pragma causes the compiler to save all necessary state at the beginning of the routine and restore it at the end of the routine. It also causes the compiler to emit the corresponding code to return from the service routine correctly.

There are two interrupt pragmas provided by the compiler for the TigerSHARC processors. The `interrupt` pragma is used to define a non-reentrant interrupt handler (one which cannot be interrupted by a subsequent interrupt). The `interrupt_reentrant` pragma is used to define a reentrant interrupt handler (one which can be interrupted by higher-priority interrupts).

To define an interrupt handler in C whose address can be put directly into the interrupt vector table, put the relevant pragma prior to the function definition. (See the example in this section.)

It is possible to use the interrupt dispatcher in the default C run-time library to register an interrupt or an exception handler and in the same application install a routine marked with the `interrupt` pragma into other entries in the vector table.

It is not possible to raise an interrupt defined in this way using the `raise` function as this is currently only provided to raise interrupts registered with the interrupt dispatcher.

Interrupt Pragma Example:
This program is designed to run on an ADSP-TS101 EZ-KIT Lite board. It flashes the **FLAG2** LED (for the given processor on which it is running). Pressing the **IRQ0** button (for the same processor) causes the rate

of flash to change. These two actions are handled by interrupts, the service routines written in C using the `interrupt` pragma. While this is happening, the main loop of the program produces some output.

```
#include <stdio.h>

#include <builtins.h>
#include <sysreg.h>
#include <defts101.h>

#define SQCTL_TMRORN MAKE_BITMASK_(SQCTL_TMRORN_P)

#pragma interrupt
void timer0h_isr (void) {
   static int led = 0;
   led = !led;
   if (led == 1) {
      __builtin_sysreg_write(__SQCTLST, SQCTL_FLAG2_OUT);
   } else {
      __builtin_sysreg_write(__SQCTLCL, ~SQCTL_FLAG2_OUT);
   }
}

#pragma interrupt
void irq0_isr (void) {
   static int speed = 0;

   speed = !speed;

   if (speed == 1) {
      __builtin_sysreg_write (__TMRIN0H, 0);
      __builtin_sysreg_write (__TMRIN0L, 250000000);
   } else {
      __builtin_sysreg_write (__TMRIN0H, 0);
      __builtin_sysreg_write (__TMRIN0L, 125000000);
   }
}

int main (void) {
   int r;
```

```
    /* Enable output on FLAG2 (to allow the LED state to
       be set on EZ-KIT hardware */
    __builtin_sysreg_write (__SQCTLST, SQCTL_FLAG2_EN);

    /* Install the two service routines into the vector table */
    __builtin_sysreg_write (__IVTIMER0HP, (int) timer0h_isr);
    __builtin_sysreg_write (__IVIRQ0, (int) irq0_isr);

    /* Set Timer 0 to count 125 million cycles */
    __builtin_sysreg_write (__TMRIN0H, 0);
    __builtin_sysreg_write (__TMRIN0L, 125000000);

    /* Make the IRQ0 interrupt edge-sensitive */
    __builtin_sysreg_write (__SQCTLCL, ~(1<<SQCTL_IRQ0_EDGE_P));

    /* Set Timer 0 running */
    __builtin_sysreg_write (__SQCTLST, (1 << SQCTL_TMR0RN_P));

    /* Set the global interrupt enable bit and the IRQ0 and
       TIMER0HP bits in the interrupt mask register */
    r = __builtin_sysreg_read (__IMASKH);
    r |= (1<<INT_GIE_P) | (1<<INT_IRQ0_P) | (1<<INT_TIMER0H_P);
    __builtin_sysreg_write (__IMASKH, r);

    /* While waiting for interrupts, produce some output */
    for (r=0; r<0x7fffffff; r++) {
       printf ("Loop %d\n", r);
    }
}
```

Loop Optimization Pragmas

Loop optimization pragmas give the compiler additional information about usage within a particular loop, which allows the compiler to perform more aggressive optimization. The pragmas are placed before the loop statement, and apply to the statement that immediately follows,

which must be a `for`, `while` or `do` statement to have effect. In general, it is most effective to apply loop pragmas to inner-most loops, since the compiler can achieve the most savings there.

#pragma all_aligned

This pragma tells the compiler that all pointer-induction variables in the loop are initially aligned to the maximum interesting alignment of the architecture. For TigerSHARC processors, that is quad-word aligned. The pragma takes an optional argument (`n`) which can specify that the pointers are aligned after `n` iterations. Therefore, the `#pragma all_aligned(1)` says that after one iteration, all the pointer induction variables of the loop are so aligned. In other words, the default argument is zero.

#pragma different_banks

This pragma allows the compiler to assume that groups of memory accesses, based on different pointers within a loop, reside in different memory banks. By scheduling them together, memory access is much improved.

#pragma loop_count(*min, max, modulo*)

This pragma appears just before the loop it describes. The pragma asserts that the loop iterates at least `min` times, no more than `max` times, and a multiple of `modulo` times. This information enables the optimizer to omit loop guards and to decide whether the loop is worth completely unrolling and whether code needs to be generated for odd iterations. The last two arguments can be omitted if they are unknown.

For example,

```
int i;
#pragma loop_count(24, 48, 8)
for (i=0; i < n; i++)
```

The `#pragma must_iterate(min, max, modulo)` is also accepted for compatibility with other compilers.

#pragma loop_unroll *N*

The `loop_unroll` pragma can be used only before a `for`, `while` or `do..while` loop. The pragma takes exactly one positive integer argument, `N`, and it instructs the compiler to unroll the loop `N` times prior to further transforming the code.

In the most general case, the effect of:

```
#pragma loop_unroll N
for ( init statements;  condition;  increment code) {
   loop_body
}
```

is equivalent to transforming the loop to:

```
for ( init statements;  condition;  increment code) {
   loop_body      /* copy 1 */
   increment_code
   if (!condition)
      break;

   loop_body      /* copy 2 */
   increment_code
   if (!condition)
      break;

      ...

   loop_body      /* copy N-1 */
   increment_code
   if (!condition)
      break;

   loop_body      /* copy N */
}
```

Similarly, the effect of

```
#pragma loop_unroll N
while ( condition ) {
   loop_body
}
```

is equivalent to transforming the loop to:

```
while ( condition ) {
   loop_body      /* copy 1 */
   if (!condition)
      break;

   loop_body      /* copy 2 */
   if (!condition)
      break;

      ...

   loop_body      /* copy N-1 */
   if (!condition)
      break;

   loop_body      /* copy N */
}
```

and the effect of:

```
#pragma loop_unroll N
do {
   loop_body
} while ( condition )
```

is equivalent to transforming the loop to:

```
do {
   loop_body      /* copy 1 */
   if (!condition)
      break;
```

```
    loop_body      /* copy 2 */
    if (!condition)
       break;

       ...

    loop_body      /* copy N-1 */
    if (!condition)
       break;

    loop_body      /* copy N */
} while ( condition )
```

#pragma no_alias

Use this pragma to tell the compiler that the following loop has no loads or stores that conflict due to references to the same location through different pointers, known as "`aliases`".

In the example,

```
void vadd(int *a, int *b, int *out, int n) {
   int i;
#pragma no_alias
   for (i=0; i < n; i++)
       out[i] = a[i] + b[i];
}
```

the use of the `no_alias` pragma just before the loop informs the compiler that the pointers `a`, `b` and `out` point to different arrays, so no load from `b` or `a` uses the same address as any store to `out`. Therefore, `a[i]` or `b[i]` is never an alias for `out[i]`.

Using the `no_alias` pragma can lead to better code because it allows the loads and stores to be reordered and any number of iterations to be performed concurrently (rather than just two at a time), thus providing better software pipelining by the optimizer.

#pragma no_vectorization

This pragma is used to turn off all vectorization for the loop on which it is specified.

#pragma vector_for

This pragma notifies the compiler that it is safe to execute some number of consecutive iterations of the loop in parallel. Therefore, the form `#pragma vector_for (number)` notifies the compiler that a *number* of consecutive iterations maybe executed in parallel. The `#pragma vector_for (without the number)` form tells the compiler that any number of consecutive iterations may be executed in parallel.

The `vector_for` pragma does not force the compiler to vectorize the loop; the optimizer checks various properties of the loop and does not vectorize it if it believes it is unsafe or if it cannot deduce that the various properties necessary for the vectorization transformation are valid.

Strictly speaking, the pragma simply disables checking for loop-carried dependencies.

```
void copy(short *a, short *b) {
int i;
#pragma vector_for
   for (i=0; i<100; i++)
       a[i] = b[i];
}
```

In cases where vectorization is impossible (for example, if array `a` is aligned on a word boundary but array `b` is not), the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

Function Side-Effect Pragmas

The function side-effect pragmas (`alloc`, `pure`, `const`, `regs_clobbered`, `overlay` and `result_alignment`) are used before a function declaration to give the compiler additional information about the function in order to enable it to improve the code surrounding the function call. These pragmas should be placed before a function declaration and should apply to that function.

For example,

```
#pragma pure
long dot(short*, short*, int);
```

#pragma alloc

This pragma tells the compiler that the function behaves like the library function "`malloc`", returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call. In the example,

```
#define N 100

#pragma alloc
int *new_buf(void);
int *vmul(int *a, int *b) {
   int *out = new_buf();
   for (i = 0; i < N; ++i)
      out[i] = a[i] * b[i];
   return out;
}
```

the compiler can reorder the iterations of the loop because the `#pragma alloc` tells it that `a` and `b` cannot overlap `out`.

The GNU attribute `malloc` is also supported with the same meaning.

#pragma const

This pragma is a more restrictive form of the `pure` pragma. It tells the compiler that the function does not read from global variables as well as not write to them or read or write volatile variables. The result is therefore a function of its parameters. If any of the parameters are pointers, the function may not read the data they point at.

#pragma noreturn

This pragma can be placed before a function prototype or definition. Its use tells the compiler that the function to which it applies will never return to its caller. For example, a function such as the standard C function "exit" never returns.

The use of this pragma allows the compiler to treat all code following a call to a function declared with the pragma as unreachable and hence removable.

```
#pragma noreturn
void func() {
   while(1);
}

main() {
   func();
   /* any code here will be removed */
}
```

#pragma pure

This pragma tells the compiler that the function does not write to any global variables, and does not read or write any volatile variables. Its result, therefore, is a function of its parameters or of global variables. If any of the parameters are pointers, the function may read the data they point at but it may not write it.

Since the function call has the same effect every time it is called (between assignments to global variables), the compiler need not generate the code for every call. Therefore, in this example,

```
#pragma pure
long sdot(short *, short *, int);

long tendots(short *a, short *b, int n) {
   int i;
   long s = 0;
   for (i = 1; i < 10; ++i)
      s += sdot(a, b, n);  // call can get hoisted out of loop
   return s;}
```

the compiler can replace the ten calls to `sdot` with a single call made before the loop.

#pragma regs_clobbered *string*

This pragma may be used with a function declaration or definition to specify which registers are modified (or clobbered) by that function. The *`string`* contains a list of registers and is case-insensitive.

When used with an external function declaration, this pragma acts as an assertion telling the compiler something it would be unable to discover for itself. In the example,

```
#pragma regs_clobbered "xr5 xr8 j6"
void f(void);
```

the compiler knows that only registers `r5`, `r8` and `j6` may be modified by the call to `f`, so it may keep local variables in other registers across that call.

The `regs_clobbered` pragma may also be used with a function definition, or a declaration preceding a definition (when it acts as a command to the compiler to generate register saves, and restores on entry and exit from the function) to ensure it only modifies the registers in string.

For example,

```
#pragma regs_clobbered "j0 j4"
int g(int a) {
   return a+3;
}
```

The `regs_clobbered` pragma may not be used in conjunction with `#pragma interrupt`. If both are specified, a warning is issued and the `regs_clobbered` pragma is ignored.

To obtain optimum results with the pragma, it is best to restrict the clobbered set to be a subset of the default scratch registers. When considering when to apply the `regs_clobbered` pragma, it may be useful to look at the output of the compiler to see how many scratch registers were used. Restricting the volatile set to these registers will produce no impact on the code produced for the function but may free up registers for the caller to allocate across the call site.

The `regs_clobbered` pragma cannot be used in any way with pointers to functions. A function pointer cannot be declared to have a customized clobber set, and it cannot take the address of a function which has a customized clobber set. The compiler raises an error if either of these actions are attempted.

String Syntax

A `regs_clobbered` string consists of a list of registers, register ranges, or register sets that are clobbered (Table 1-21). The list is separated by spaces, commas, or semicolons.

A *`register`* is a single register name, which is the same as that which may be used in an assembly file. Register names (when not used in ranges) may be pairs, quads or SIMD. Therefore, "`r1`", "`xyr2`", "`yr3:0`" and "`r3:0`" are acceptable register names specifying register sets "`xr1,yr1`", "`xr2,yr2`", "`yr0,yr1,yr2,yr3`" and "`xr0,xr1,xr2,xr3,yr0,yr1,yr2,yr3`", respectively.

A *register range* consists of start and end registers which both reside in the same register class, separated by a hyphen. All registers between the two (inclusive) are clobbered.

A *register set* is a name for a specific set of commonly clobbered registers that is predefined by the compiler. Table 1-21 shows defined clobbered register sets,

Table 1-21. Clobbered Register Sets

Set	Registers
CCset	XSTAT, YSTAT, JSTAT, KSTAT
MRset	XMR0-XMR4, YMR0-YMR4
PRset	XPR0, XPR1, YPR0, YPR1
LCset	LC0, LC1
ACCELset	XTR0-XTR31, YTR0-YTR31, XTHR0-XTHR3, YTHR0-YTHR3
DABset	XDAB0-XDAB3, YDAB0-YDAB3
CBset	J0-J3, K0-K3, JL0-JL3, KL0-KL3, JB0-JB3, KB0-KB3
Xscratch	Members of X registers that are scratch by default, XSTAT
Yscratch	Members of Y registers that are scratch by default, YSTAT
XYscratch	Xscratch , Yscratch
Jscratch	Members of J registers that are scratch by default, JSTAT
Kscratch	Members of K registers that are scratch by default, KSTAT
JKscratch	Jscratch, Kscratch
XYJKscratch	JKscratch, XYscratch
ALLscratch	Entire default volatile set

The strings are also valid as GNU asm clobbered sets.

When the compiler detects an illegal string, a warning is issued and the default volatile set as defined in this compiler manual is used instead.

Unclobberable and Must Clobber Registers
There are certain caveats as to what registers may or must be placed in the clobbered set. On TigerSHARC processors, the registers `J26`, `J27`, `K26` and `K27` may not be specified in the clobbered set, as the correct operation of the function call requires their values to be preserved. If the user specifies them in the clobbered set, a warning is issued and these registers are removed from the specified clobbered set.

Registers from these classes,

```
X, Y, J, K, XT, YT, XTH, YTH, XDAB, YDAB, LC, JB, KB, JL, KL,
XMR, YMR, XP, YP, XSTAT, YSTAT, JSTAT, KSTAT
```

may be specified in the clobbered set and code is generated to save them as necessary. All other registers are never preserved whether specified as clobbered or not. They are not automatically used by the compiler. It is assumed that the users, should they change the register, would like this change to be preserved.

Function Return Registers
Function return registers are visible to the caller. They may or may not appear in the clobbered set of the callee. (It makes no difference to the generated code; the return register is never saved and restored.) Only the return register used by the particular function return type is special. Return registers used by different return types are treated in the clobbered list in the conventional way. For example,

```
int f();
/* returns via J8. XR8 and XR9 may be preserved across call */
long long f();
/* returns via XR9:8. J8 may be preserved across call     */
void f();
/* J8, XR8 and XR9 may all be preserved across the call   */
```

Function Parameters
Function calling conventions are visible to the caller and do not affect the clobbered set that may be used on a function. For example,

```
#pragma regs_clobbered ""   // clobbers nothing
void f(int a, int b);
void g() {
     f(2,3);
}
```

The parameters `a` and `b` are passed in registers registers `J4` and `J5` respectively. No matter what happens in function *f*, the values of `J4` and `J5` after the call are still set to 2 and 3, respectively.

Pragma Interrupt Restriction
The `regs_clobbered` pragma may not be used in conjunction with `#pragma interrupt`. If both are specified, a warning is issued and the `regs_clobbered` pragma is ignored.

#pragma regs_clobbered_call *string*

This pragma may be applied to a statement to indicate that the call within the statement uses a modified volatile register set. The pragma is closely related to `#pragma regs_clobbered`, but avoids some of the restrictions that relate to that pragma.

These restrictions arise because the `regs_clobbered` pragma applies to a function's declaration—when the call is made, the clobber set is retrieved from the declaration automatically. This is not possible when the declaration is not available, because the function being called is not directly tied to a declaration of a specific function. This affects:

- pointers to functions
- class methods
- pointers to class methods
- virtual functions

In such cases, the `regs_clobbered_call` pragma can be used at the call site to inform the compiler directly of the volatile register set to be used during the call.

The pragma's syntax is as follows:

```
#pragma regs_clobbered_call clobber_string
 statement
```

where *`clobber_string`* follows the same format as for the `regs_clobbered` pragma and *`statement`* is the C statement containing the call expression.

There must be only a single call within the statement; otherwise, the statement is ambiguous.

For example,

```
#pragma regs_clobbered "xr0 xr1 xr2"
#int func(int arg) { /* some code */ }

int (*fnptr)(int) = func;

int caller(int value) {
  int r;

#pragma regs_clobbered_call "xr0 xr1 xr2"
  r = (*fnptr)(value);
  return r;
}
```

When you use the `regs_clobbered_call` pragma, you must ensure that the called function does indeed only modify the registers listed in the clobber set for the call—the compiler does not check this for you. It is valid for the callee to clobber less than is listed in the call's clobber set. It is also valid for the callee to modify registers outside of the call's clobber set, as long as the callee saves the values first and restores them before returning to the caller.

The following examples show this.

Example 1:

```
#pragma regs_clobbered "xr0 xr1 xr2"
void callee(void) { ... }

#pragma regs_clobbered_call "xr0 xr1 xr2"
callee();       // Okay - clobber sets match
```

Example 2:

```
#pragma regs_clobbered "xr0 xr1"
void callee(void) { ... }

#pragma regs_clobbered_call "xr0 xr1 xr2"
callee();       // Okay - callee clobber set is a subset
                // of call's set
```

Example 3:

```
#pragma regs_clobbered "xr0 xr1 xr2"
void callee(void) { ... }

#pragma regs_clobbered_call "xr0 xr1"
callee();       // Error - callee clobbers more than
                // indicated by call.
```

Example 4:

```
void callee(void) { ... }

#pragma regs_clobbered_call "xr0 xr1 xr2"
callee();       // Error - callee uses default set larger
                // than indicated by call.
```

Limitations
Pragma `regs_clobbered_call` may not be used on constructors or destructors of C++ classes.

The pragma only applies to the call in the immediately-following statement. If the immediately-following line contains more than one statement, the pragma only applies to the first statement on the line:

```
#pragma regs_clobbered "xr0 xr1 xr2"
x = foo(); y = bar();   // only "x = foo();" is affected by
                        // the pragma.
```

Similarly, if the immediately-following line is a sequence of declarations that use calls to initialize the variables, then only the first declaration is affected:

```
#pragma regs_clobbered "xr0 xr1 xr2"
int x = foo(), y = bar();   // only "x = foo()" is affected
                            // by the pragma.
```

Moreover, if the declaration with the call-based initializer is not the first in the declaration list, the pragma will have no effect:

```
#pragma regs_clobbered "xr0 xr1 xr2"
int w = 4, x = foo(); y = bar();   // pragma has no effect
                                   // on “w = 4”.
```

The pragma has no effect on function calls that get inlined. Once a function call is inlined, the inlined code obeys the clobber set of the function into which it has been inlined. It does not continue to obey the clobber set that will be used if an out-of-line copy is required.

#pragma overlay

When compiling code which involves one function calling another in the same source file, the compiler optimizer can propagate register information between the functions. This means that it can record which scratch registers are clobbered over the function call. This can cause problems when compiling overlaid functions, as the compiler may assume that certain scratch registers are not clobbered over the function call, but they are

clobbered by the overlay manager. `#pragma overlay`, when placed on the definition of a function, will disable this propagation of register information to the function's callers. For example:

```
#pragma overlay
int add(int a, int b)
{
    // callers of function add() assume it clobbers
    // all scratch registers
    return a+b;
}
```

#pragma result_alignment (*n*)

This pragma asserts that the pointer or integer returned by the function has a value that is a multiple of *n*.

The pragma is often used in conjunction with the `#pragma alloc` of custom-allocation functions that return pointers that are more strictly aligned than could be deduced from their type.

General Optimization Pragmas

The compiler supports several pragmas which can change the optimization level while a given module is being compiled. These pragmas must be used globally, immediately prior to a function definition. The pragmas do not just apply to the immediately following function; they remain in effect until the end of the compilation, or until superseded by one of the following `optimize_` pragmas.

- **#pragma optimize_off**
  This pragma turns off the optimizer, if it was enabled, meaning it has the same effect as compiling with no optimization enabled.
- **#pragma optimize_for_space**

This pragma turns the optimizer back on, if it was disabled, or sets the focus to give reduced code size a higher priority than high performance, where these conflict.

- **#pragma optimize_for_speed**
  This pragma turns the optimizer back on, if it was disabled, or sets the focus to give high performance a higher priority than reduced code size, where these conflict.

- **#pragma optimize_as_cmd_line**
  This pragma resets the optimization settings to be those specified on the `ccts` command line when the compiler was invoked.

These are code examples for the `optimize_` pragmas.

```
#pragma optimize_off
void non_op() { /* non-optimized code */ }

#pragma optimize_for_space
void op_for_si() { /* code optimized for size */ }

#pragma optimize_for_speed
void op_for_sp() { /* code optimized for speed */ }
/* subsequent functions declarations optimized for speed */
```

- **#pragma separate_mem_segments (var1, var2)**
  The `separate_mem_segments` pragma specifies that the two variables `var1` and `var2` should be placed into different memory segments. Refer to Chapter 2 of the *VisualDSP++ 4.5 Linker and Utilities Manual* for more information.

Inline Control Pragmas

The compiler supports two pragmas to control the inlining of code. These pragmas are `#pragma always_inline` and `#pragma never_inline`.

#pragma always_inline

This pragma may be applied to a function definition to indicate to the compiler that the function should always be inlined, and never called "out of line". The pragma may only be applied to function definitions with the `inline` qualifier, and may not be used on functions with variable-length argument lists. It is invalid for function definitions that have interrupt-related pragmas associated with them.

If the function in question has its address taken, the compiler cannot guarantee that all calls are inlined, so a warning is issued.

See "Function Inlining" on page 1-95 for details of pragma precedence during inlining.

The following are examples of the `always_inline` pragma.

```
int func1(int a) {          // only consider inlining
  return a + 1;             // if -Oa switch is on
}

inline int func2(int b) {   // probably inlined, if optimizing
  return b + 2;
}

#pragma always_inline
inline int func3(int c) {   // always inline, even unoptimized
  return c + 3;
}

#pragma always_inline
int func4(int d) {   // error: not an inline function
  return d + 4;
}
```

#pragma never_inline

This pragma may be applied to a function definition to indicate to the compiler that function should always be called “out of line”, and that the function’s body should never be inlined.

This pragma may not be used on function definitions that have the `inline` qualifier.

See “Function Inlining” on page 1-95 for details of pragma precedence during inlining.

These are code examples for the `never_inline` pragma.

```
#pragma never_inline
int func5(int e) {   // never inlined, even with -Oa switch
  return e + 5;
}

#pragma never_inline
inline int func5(int f) {   // error: inline function
  return f + 6;
}
```

Linking Control Pragmas

Linking pragmas (`linkage_name`, `core`, `section` and `weak_entry`) change how a given global function or variable is viewed during the linking stage.

#pragma linkage_name *identifier*

This pragma associates the *`identifier`* with the next external function declaration. It ensures that the *`identifier`* is used as the external reference, instead of following the compiler’s usual conventions. If the *`identifier`* is not a valid function name, as could be used in normal function definitions, the compiler generates an error. See also the `asm` keyword (described on page 1-244).

The following example shows how to use this pragma.

```
#pragma linkage_name realfuncname
void funcname ();
void func() {
  funcname();  /* compiler generates a call to realfuncname */
}
```

#pragma core

When building a project that targets multiple processors or multiple cores on a processor, a link stage may produce executables for more than one core or processor. The interprocedural analysis (IPA) framework requires that some conventions be adhered to in order to successfully perform its analyses for such projects.

Because the IPA framework collects information about the whole program, including information on references which may be to definitions outside the current translation unit, the IPA framework must be able to distinguish these definitions and their references without ambiguity.

If any confusion were allowed about which definition a reference refers to, then the IPA framework could potentially cause bad code to be generated, or could cause translation units in the project to be continually recompiled ad infinitum. It is the global symbols that are really relevant in this respect. The IPA framework will correctly handle locals and static symbols because multiple definitions are not possible within the same file, so there can be no ambiguity.

In order to disambiguate all references and the definitions to which they refer, it is necessary to have a unique name for each definition within a given project. It is illegal to define two different functions or variables with the same name. This is illegal in single-core projects because this would lead to multiple definitions of a symbol and the link would fail. In multi-core projects, however, it may be possible to link a project with multiple definitions because one definition could be linked into each link project, resulting in a valid link. Without detailed knowledge of what

actions the linker had performed, however, the IPA framework would not be able disambiguate such multiple definitions. For this reason, to use the IPA framework, it is up to you to ensure unique names even in projects targeting multiple cores or processors.

There are a few cases for which it is not possible to ensure unique names in multi-core or multi-processor projects. One such case is `main`. Each processor or core will have its own `main` function, and these need to be disambiguated for the IPA framework to be able to function correctly. Another case is where a library (or the C run-time startup) references a symbol which the user may wish to define differently for each core.

For this reason, a compiler pragma is provided:

```
#pragma core(corename)
```

This can be provided immediately prior to a definition or a declaration. This pragma allows you to give a unique identifier to each definition. It also allows you to indicate to which definition each reference refers. The IPA framework will use this core identifier to distinguish all instances of symbols with the same name and will therefore be able to carry out its analyses correctly.

Note that the *corename* specified should only consist of alphanumeric characters. Also note that the *corename* is case sensitive.

The pragma should be used:

- On every definition (not in a library) for which there needs to be a distinct definition for each core.
- On every declaration of a symbol (not in a library) for which the relevant definition includes the use of `#pragma core`. The core specified for a declaration must agree with the core specified for the definition.

It should be noted that the IPA framework will not need to be informed of any distinction if there are two identical copies of the same function or data with the same name. Functions or data that come from objects and that are duplicated in memory local to each core, for example, will not need to be distinguished. The IPA framework does not need to know exactly which instance each reference will get linked to because the information processed by the framework is identical for each copy. Essentially, the pragma only needs to be specified on items where there will be different functions or data with the same name incorporated into the executable for each core.

Here is an example of `#pragma core` usage to distinguish two different main functions:

```
/* foo.c */
#pragma core("coreA")
int main(void) {
    /* Code to be executed by core A */
}
/* bar.c */
#pragma core("coreB")
int main(void) {
    /* Code to be executed by core B */
}
```

Omitting either instance of the pragma will cause the IPA framework to issue a fatal error indicating that the pragma has been omitted on at least one definition.

Here is an example that will cause an error to be issued because the name contains a non-alphanumeric character:

```
#pragma core("core/A")
int main(void) {
    /* Code to executed on core A */
}
```

Here is an example where the pragma needs to be specified on a declaration as well as the definitions. There is a library which contains a reference to a symbol which is expected to be defined for each core. Two more modules define the `main` functions for the two cores. Two further modules, each only used by one of the cores, makes a reference to this symbol, and therefore requires use of the pragma:

```
/* libc.c */
#include <stdio.h>
extern int core_number;
void print_core_number(void) {
    printf("Core %d\n", core_number);
}
/* maina.c */
extern void fooa(void)
#pragma core("coreA")
int core_number = 1;
#pragma core("coreA")
int main(void) {
    /* Code to be executed by core A */
    print_core_number();
    fooa();
}
/* mainb.c */
extern void foob(void)
#pragma core("coreB")
int core_number = 2;
#pragma core("coreB")
int main(void) {
    /* Code to be executed by core B */
    print_core_number();
    foob();
}
/* fooa.c */
#include <stdio.h>
#pragma core("coreA")
extern int core_number;
void fooa(void) {
    printf("Core: is core%c\n", 'A' - 1 + core_number);
```

```
}
/* foob.c */
#include <stdio.h>
#pragma core("coreB")
extern int core_number;
void fooa(void) {
    printf("Core: is core%c\n", 'A' - 1 + core_number);
}
```

In general, it will only be necessary to use `#pragma core` in this manner when there is a reference from outside the application (in a library, for example) where there is expected to be a distinct definition provided for each core, and where there are other modules that also require access to their respective definition. Notice also that the declaration of `core_number` in `lib.c` does not require use of the pragma because it is part of a translation unit to be included in a library.

A project that includes more than one definition of `main` will undergo some extra checking to catch problems that would otherwise occur in the IPA framework. For any non-template symbol that has more than one definition, the tool chain will fault any definitions that are outside libraries that do not specify a core name with the pragma. This check does not affect the normal behavior of the prelinker with respect to templates and in particular the resolution of multiple template instantiations.

To clarify:

Inside a library, `#pragma core` is not required on declarations or definitions of symbols that are defined more than once. However, a library can be responsible for forcing the application to define a symbol more than once (that is, once for each core). In this case, the definitions and declarations require the pragma to be used outside the library to distinguish the multiple instances.

It should be noted that the tool chain cannot check that uses of `#pragma core` are consistent. If you use the pragma inconsistently or ambiguously, then the IPA framework may end up causing incorrect code to be generated or causing continual recompilation of the application's files.

It is also important to note that the pragma does not change the linkage name of the symbol it is applied to in any way.

For more information on IPA, see "Interprocedural Analysis" on page 1-84.

#pragma section/#pragma default_section

The section pragmas provide greater control over the sections in which the compiler places symbols.

The `section(`*`SECTSTRING`* `[,` *`QUALIFIER`*`, ...])` pragma is used to override the target section for any global or static symbol immediately following it. The pragma allows greater control over section qualifiers compared to the `section` keyword.

The `default_section(`*`SECTKIND`* `[,` *`SECTSTRING`* `[,` *`QUALIFIER`*`, ...]])` pragma is used to override the default sections in which the compiler is placing its symbols. The default sections fall into five different categories (listed under *`SECTKIND`*), and this pragma remains in force for a section category until its next use with that particular category. The omission of a section name results in the default section being reset to be the section that was in use at the start of processing.

`SECTKIND` can be one of the following keywords:

Table 1-22. Keyword Possibilities for SECTKIND

Keyword	Description
`CODE`	Section is used to contain procedures and functions
`ALLDATA`	Section is used to contain any data (normal, read-only and uninitialized)

Table 1-22. Keyword Possibilities for SECTKIND

Keyword	Description
DATA	Section is used to contain “normal data”
CONSTDATA	Section is used to contain read-only data
BSZ	Section is used to contain uninitialized data
SWITCH	Section is used to contain jump-tables to implement C/C++ switch statements
VTABLE	Section is used to contain C++ virtual-function tables
STI	Section is used to contain C++ constructors and destructors

SECTSTRING is the double-quoted string containing the section name, exactly as it will appear in the assembler file.

QUALIFIER can be one of the following keywords:

Table 1-23. Keyword Possibilities for QUALIFIER

Keyword	Description
PM	Section is located in program memory
DM	Section is located in data memory
ZERO_INIT	Section is zero-initialized at program startup
NO_INIT	Section is not initialized at program startup
RUNTIME_INIT	Section is user-initialized at program startup
DOUBLE32	Section may contain 32-bit but not 64-bit doubles
DOUBLE64	Section may contain 64-bit but not 32-bit doubles
DOUBLEANY	Section may contain either 32-bit or 64-bit doubles
CHAR8	Section may contain 8-bit but not 32-bit chars
CHAR32	Section may contain 32-bit but not 8-bit chars .
CHARANY	Section may contain either 8-bit or 32-bit chars.

There may be any number of comma-separated section qualifiers within such pragmas, but they must not conflict with one another. Qualifiers must also be consistent across pragmas for identical section names, and omission of qualifiers is not allowed even if at least one such qualifier has appeared in a previous pragma for the same section. If any qualifiers have not been specified for a particular section by the end of the translation unit, the compiler uses default qualifiers appropriate for the target processor. The compiler always tries to honor the `section` pragma as its highest priority, and the `default_section` pragma is always the lowest priority.

For example, the following code results in the function `f` being placed in the section `foo`:

```
#pragma default_section(CODE, "bar")
#pragma section("foo")
void f() {}
```

The following code results in `x` being placed in section `zeromem`:

```
#pragma default_section(BSZ, "zeromem")
int x;
```

However, the following example does not result in the variable a being placed in section `onion` because it was declared with the `__pm` qualifier and therefore is placed in the `PM` data section:

```
#pragma default_section(DATA, "onion")
__pm int a = 4;
```

#pragma file_attr (***name***[=***value***] [, ***name***[=***value***] [...]])

This pragma directs the compiler to emit the specified attributes when it compiles a file containing the pragma. Multiple `#pragma file_attr` directives are allowed in one file.

If =*value* is omitted, the default value of "1" will be used.

The value of an attribute is all the characters after the '=' symbol and before the closing '"' symbol, including spaces. A warning will be emitted by the compiler if you have a preceding or trailing space as an attribute value, as this is likely to be a mistake.

See "File Attributes" on page 1-293 for more information on using attributes.

#pragma weak_entry

This pragma may be used before a static variable or function declaration or definition. It applies to the function/variable declaration or definition that immediately follows the pragma. Use of this pragma causes the compiler to generate the function or variable definition with weak linkage.

The following are example uses of the `pragma weak_entry` directive.

```
#pragma weak_entry
int w_var = 0;

#pragma weak_entry
void w_func(){}
```

When a symbol definition is weak, it may be discarded by the linker in favor of another definition of the same symbol. Therefore, if any modules in the application make use of the `weak_entry` pragma, interprocedural analysis is disabled becauses it would be unsafe for the compiler to predict which definition will be selected by the linker. For more information, see "Interprocedural Analysis" on page 1-84.

Class Conversion Optimization Pragmas

The class conversion optimization pragmas (`param_never_null`, `suppress_null_check`) allow the compiler to generate more efficient code when converting class pointers from a pointer-to-derived-class to a pointer-to-base-class, by asserting that the pointer to be converted will never be a null pointer. This allows the compiler to omit the null check during conversion.

#pragma param_never_null ***param_name [...]***

This pragma must immediately precede a function definition. It specifies a name or a list of space-separated names, which must correspond to the parameter names declared in the function definition. It checks that the named parameter is a class pointer type. Using this information it will generate more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer during the conversion. For example,

```
#include <iostream>
using namespace std;
class A {
    int a;
};
class B {
    int b;
};
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

#pragma param_never_null pc
void func(C *pc)
{
```

```
    B *pb;
    pb = pc;  /* without pragma the code generated has to
                 check for NULL */
    if (pb != bpart)
        fail = true;
}

int main(void)
{
    func(&obj);
    if (fail)
        cout << “Test failed” << endl;
    else
        count << “Test passed” << endl;
    return 0;
}
```

#pragma suppress_null_check

This pragma must immediately precede an assignment of two pointers or a declaration list.

If the pragma precedes an assignment it indicates that the second operand pointer is not null and generates more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer before assignment.

On a declaration list it marks all variables as not being the null pointer. If the declaration contains an initialization expression, that expression is not checked for null.

```
#include <iostream>
using namespace std;
class A {
    int a;
};
class B {
    int b;
};
```

```
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

void func(C *pc)
{
    B *pb;
    #pragma suppress_null_check
    pb = pc;  /* without pragma the code generated has to
                 check for NULL */
    if (pb != bpart)
        fail = true;
}

void func2(C *pc)
{
    #pragma suppress_null_check
    B *pb = pc, *pb2 = pc;  /* pragma means these initializations
                 need not check for NULL. It also marks pb and pb2
                 as never being NULL, so the compiler will not
                 generate NULL checks in class conversions using
                 these pointers. */
    if (pb != bpart || pb2 != bpart)
        fail = true;
}

int main(void)
{
    func(&obj);
    func2(&obj);
    if (fail)
        cout << “Test failed” << endl;
    else
        count << “Test passed” << endl;
    return 0;
}
```

Template Instantiation Pragmas

The template instantiation pragmas (`instantiate`, `do_not_instantiate` and `can_instantiate`) give fine-grain control over where (that is, in which object file) the individual instances of template functions, member functions, and static members of template classes are created. The creation of these instances from a template is known in "C++ speak" as instantiation. As templates are a feature of C++, these pragmas are allowed only in `-c++` mode.

Refer to "Compiler C++ Template Support" on page 1-289 for more information on how the compiler handles templates.

The instantiation pragmas take the name of an instance as a parameter, as shown in Table 1-24.

Table 1-24. Instance Names

Name	Parameter
a template class name	`A<int>`
a template class declaration	`class A<int>`
a member function name	`A<int>::f`
a static data member name	`A<int>::I`
a static data declaration	`int A<int>::I`
a member function declaration	`void A<int>::f(int, char)`
a template function declaration	`char* f(int, float)`

If the instantiation pragmas are not used, the compiler selects object files where all required instances automatically instantiate during the pre-linking process.

#pragma instantiate *instance*

This pragma requests the compiler to instantiate *instance* in the current compilation. For example,

```
#pragma instantiate class Stack<int>
```

causes all static members and member functions for the `int` instance of a template class `Stack` to be instantiated, whether they are required in this compilation or not.

The example,

```
#pragma instantiate void Stack<int>::push(int)
```

causes only the individual member function `Stack<int>::push(int)` to be instantiated.

#pragma do_not_instantiate *instance*

This pragma directs the compiler not to instantiate *instance* in the current compilation. For example,

```
#pragma do_not_instantiate int Stack<float>::use_count
```

prevents the compiler from instantiating the static data member `Stack<float>::use_count` in the current compilation.

#pragma can_instantiate *instance*

This pragma tells the compiler that if instance is required anywhere in the program, it should be instantiated in this compilation.

Currently, this pragma forces the instantiation even if it is not required anywhere in the program. Therefore, it has the same effect as `#pragma instantiate`.

Header File Control Pragmas

The header file control pragmas (`hdrstop`, `no_implicit_inclusion`, `no_pch`, `once`, and `system_header`) help the compiler to handle header files.

#pragma hdrstop

This pragma is used with the `-pch` (precompiled header) switch (on page 1-54). The switch tells the compiler to look for a precompiled header (`.pch` file), and, if it cannot find one, to generate a file for use on a later compilation. The `.pch` file contains a snapshot of all the code preceding the header stop point.

By default, the header stop point is the first non-preprocessing token in the primary source file. The `#pragma hdrstop` can be used to set the point earlier in the source file.

In the example,

```
#include "standard_defs.h"
#include "common_data.h"
#include "frequently_changing_data.h"

int i;
```

the default header stop point is start of the declaration of `i`. This might not be a good choice, as in this example, "`frequently_changing_data.h`" might change frequently, causing the `.pch` file to be regenerated often, and, therefore, losing the benefit of precompiled headers. The `hdrstop` pragma can be used to move the header stop to a more appropriate place.

For the following example,

```
#include "standard_defs.h"
#include "common_data.h"
#pragma hdrstop
#include "frequently_changing_data.h"
```

```
int i;
```

the precompiled header file would not include the contents of `frequently_changing_data.h`, as it is included after the `hdrstop` pragma, and so the precompiled header file would not need to be regenerated each time f`requently_changing_data.h` was modified.

#pragma no_implicit_inclusion

With the `-c++` switch (on page 1-22), for each included `.h` file, the compiler attempts to include the corresponding `.c` or `.cpp` file. This is called implicit inclusion.

If `#pragma no_implicit_inclusion` is placed in a `.h` file, the compiler does not implicitly include the corresponding `.c` or `.cpp` file with the `-c++` switch. This behavior only affects the `.h` file with `#pragma no_implicit_inclusion` within it and the corresponding `.c` or `.cpp` files.

For example, if there are the following files,

`t.c` containing

```
#include “m.h”
```

and `m.h` and `m.c` are both empty, then

```
ccts -c++ t.c -M
```

shows the following dependencies for `t.c`:

```
t.doj: t.c
t.doj: m.h
t.doj: m.C
```

If the following line is added to `m.h`,

```
#pragma no_implicit_inclusion
```

running the compiler as before would not show `m.c` in the dependencies list, such as:

```
t.doj: t.c
t.doj: m.h
```

#pragma no_pch

This pragma overrides the `-pch` switch (on page 1-54) for a particular source file. It directs the compiler not to look for a `.pch` file and not to generate one for the specified source file.

#pragma once

This pragma, which should appear at the beginning of a header file, tells the compiler that the header is written in such a way that including it several times has the same effect as including it once. For example,

```
#pragma once
#ifndef FILE_H
#define FILE_H
... contents of header file ...
#endif
```

In this example, the `#pragma once` is actually optional because the compiler recognizes the `#ifndef/#define/#endif` idiom and does not reopen a header that uses it.

#pragma system_header

This pragma identifies an include file as a file supplied with VisualDSP++. The VisualDSP++ compiler makes use of this information to help optimize uses of the supplied library functions and inline functions that these files define. The pragma should not be used in user application source.

Diagnostic Control Pragmas

The compiler supports `#pragma diag(`*`action: diag`* `[, `*`diag`* `...])` which allows selective modification of the severity of compiler diagnostic messages.

The directive has three forms:

- modify the severity of specific diagnostics
- modify the behavior of an entire class of diagnostics
- save or restore the current behavior of all diagnostics

Modifying the Severity of Specific Diagnostics

This form of the directive has the following syntax:

```
#pragma diag(ACTION: DIAG [, DIAG ...])
```

The *action:* qualifier can be one of the following keywords:

Table 1-25. Keywords for *action:* Qualifier

Keyword	Action
suppress	Suppresses all instances of the diagnostic
remark	Changes the severity of the diagnostic to a remark.
warning	Changes the severity of the diagnostic to a warning.
error	Changes the severity of the diagnostic to an error.
restore	Restores the severity of the diagnostic to what it was originally at the start of compilation after all command-line options were processed.

The *diag* qualifier can be one or more comma-separated compiler diagnostic numbers without the preceding "cc" (but can include leading zeros). The choice of error numbers is limited to those that may have their severity overridden (such as those that are displayed with a "{D}" in the error message). In addition, those diagnostics that are emitted by the compiler backend (for example, after lexical analysis and parsing) cannot have their severity overridden either. Any attempt to override diagnostics that may not have their severity changed is silently ignored.

Modifying the Behavior of an Entire Class of Diagnostics

This form of the directive has the following syntax:

```
#pragma diag(ACTION)
```

The effects are as follows:

- **#pragma diag*(errors)***

  This pragma can be used to inhibit all subsequent warnings and remarks (equivalent to the `-w` switch option).

- **#pragma diag*(remarks)***

  This pragma can be used to enable all subsequent remarks and warnings (equivalent to the `-Wremarks` switch option)

- **#pragma diag(*warnings*)**

  This pragma can be used to restore the default behavior when neither `-w` or `-Wremarks` is specified, which is to display warnings but inhibit remarks.

Saving or Restoring the Current Behavior of All Diagnostics

This form has the following syntax:

```
#pragma diag(ACTION)
```

The effects are as follows:

- **#pragma diag(push)**

  This pragma may be used to store the current state of the severity of all diagnostic error messages.

- **#pragma diag(pop)**

  This pragma restores all diagnostic error messages that was previously saved with the most recent `push`.

All `#pragma diag(push)` directives must be matched with the same number of `#pragma diag(pop)` directives in the overall translation unit, but need not be matched within individual source files. Note that the error threshold (set by the `remarks`, `warnings` or `errors` keywords) is also saved and restored with these directives.

The duration of such modifications to diagnostic severity are from the next line following the pragma to either the end of the translation unit, the next `#pragma diag(pop)` directive, or the next overriding `#pragma diag()` directive with the same error number. These pragmas may be used anywhere and are not affected by normal scoping rules.

All command-line overrides to diagnostic severity are processed first and any subsequent `#pragma diag()` directives will take precedence, with the restore action changing the severity back to that at the start of compilation after processing the command-line switch overrides.

Note that the directives to modify specific diagnostics are singular. (For example, "`error`"), and the directives to modify classes of diagnostics are plural (["`errors`"].)

Memory Bank Pragmas

The memory bank pragmas provide additional performance characteristics for the memory areas used to hold code and data for the function.

By default, the compiler assumes that there are no external costs associated with memory accesses. This strategy allows optimal performance when the code and data are placed into high-performance internal memory. In cases where the performance characteristics of memory are known in advance, the compiler can exploit this knowledge to improve the scheduling of generated code.

Note that memory banks are different from sections:

- Section is a "hard" placement, using a name that is meaningful to the linker. If the `.LDF` file does not map the named section, a linker error occurs.
- A memory bank is a "soft" placement, using a name that is not visible to the linker. The compiler uses optimization to take advantage of the bank's performance characteristics. However, if the `.LDF` file maps the code or data to memory that performs differently, the application still functions (albeit with a possible reduction in performance).

#pragma code_bank(*bankname*)

This pragma informs the compiler that the instructions for the immediately-following function are placed in a memory bank called *`bankname`*. Without this pragma, the compiler assumes that the instructions are placed into a bank called "`__code`". When optimizing the function, the compiler takes note of attributes of memory bank *`bankname`*, and determines how long it takes to fetch each instruction from the memory bank.

In the example,

```
#pragma code_bank(slowmem)
int add_slowly(int x, int y) { return x + y; }

int add_quickly(int a, int b) { return a + b; }
```

the `add_slowly()` function is placed into the bank "`slowmem`", which may have different performance characteristics from the "`__code`" bank, into which `add_quickly()` is placed.

#pragma data_bank(*bankname*)

This pragma informs the compiler that the immediately-following function uses the memory bank *bankname* as the model for memory accesses for non-local data that does not otherwise specify a memory bank. Without this pragma, the compiler assumes that non-local data should use the bank "__data" for behavioral characteristics.

In the example,

```
#pragma data_bank(green)
int green_func(void)
{
      extern int arr1[32];
      extern int bank("blue") i;
      i &= 31;
      return arr1[i++];
}
int blue_func(void)
{
      extern int arr2[32];
      extern int bank("blue") i;
      i &= 31;
   return arr2[i++];
}
```

In both `green_func()` and `blue_func()`, `i` is associated with the memory bank "`blue`", and the retrieval and update of `i` are optimized to use the performance characteristics associated with memory bank "`blue`".

The array `arr1` does not have an explicit memory bank in its declaration. Therefore, it is associated with the memory bank "`green`", because `green_func()` has a specific default data bank. In contrast, `arr2` is associated with the memory bank "__data", because `blue_func()` does not have a `#pragma data_bank` preceding it.

#pragma stack_bank(*bankname*)

This pragma informs the compiler that all locals for the immediately-following function are to be associated with memory bank *bankname*, unless they explicitly identify a different memory bank. Without this pragma, all locals are assumed to be associated with the memory bank "__stack". In the example,

```
#pragma stack_bank(mystack)
short dotprod(int n, const short *x, const short *y)
{
      int sum = 0;
      int i = 0;
      for (i = 0; i < n; i++)
          sum += *x++ * *y++;
      return sum;
}

int fib(int n)
{
      int r;
      if (n < 2) {
         r = 1;
      } else {
         int a = fib(n-1);
         int b = fib(n-2);
         r = a + b;
      }
      return r;
}

#pragma interrupt
#pragma stack_bank(sysstack)
void count_ticks(void)
{
      extern int ticks;
      ticks++;
}
```

The `dotprod()` function places the `sum` and `i` values into the memory bank "`mystack`", while `fib()` places `r`, `a` and `b` into the memory bank "`__stack`", because there is no `stack_bank` pragma. The `count_ticks()` function does not declare any local data, but any compiler-generated local storage make use of the "`sysstack`" memory bank's performance characteristics.

#pragma bank_memory_kind(*bankname, kind*)

This pragma informs the compiler what kind of memory the memory bank *`bankname`* is. The following kinds of memory are allowed by the compiler:

- internal – the memory bank is high-speed in-core memory
- external – the memory bank is external to the processor

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

In the example,

```
#pragma bank_memory_kind(blue, internal)
int sum_list(bank("blue") const int *data, int n)
{
       int sum = 0;
       while (n--)
           sum += data[n];
       return sum;
}
```

the compiler knows that all accesses to the `data[]` array are to the "`blue`" memory bank, and hence to internal, in-core memory.

#pragma bank_read_cycles(*bankname, cycles*)

This pragma tells the compiler that each read operation on the memory bank *bankname* requires the *cycles* cycles before the resulting data is available. This allows the compiler to schedule sufficient code between the initiation of the read and the use of its results, to prevent unnecessary stalls.

In the example,

```
#pragma bank_read_cycles(slowmem, 20)
int dotprod(int n, const int *x, bank("slowmem") const int *y)
{
      int i, sum;
      for (i=sum=0; i < n; i++)
          sum += *x++ * *y++;
      return sum;
}
```

the compiler assumes that a read from `*x` takes a single cycle, as this is the default read time, but that a read from `*y` takes twenty cycles, because of the pragma.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

#pragma bank_write_cycles(*bankname, cycles*)

This pragma tells the compiler that each write operation on memory bank *bankname* requires the *cycles* cycles before it completes. This allows the compiler to schedule sufficient code between the initiation of the write and a subsequent read or write to the same location, to prevent unnecessary stalls.

In the following example,

```
void write_buf(int n, const char *buf)
{
      volatile bank("output") char *ptr = REG_ADDR;
```

```
        while (n--)
            *ptr = *buf++;
}
#pragma bank_write_cycles(output, 6)
```

the compiler knows that each write through `ptr` to the "`output`" memory bank takes six cycles to complete.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition. This is shown in the preceding example.

#pragma bank_optimal_width(*bankname, width*)

This pragma informs the compiler that *`width`* is the optimal number of bits to transfer to/from memory bank *`bankname`* in a single cycle. This can be used to indicate to the compiler that some memories can benefit from vectorization and similar strategies more than others. The *`width`* parameter must be 8, 16, 24 or 32.

In the example,

```
void memcpy_simple(char *dst, const char *src, size_t n)
{
        while (n--)
            *dst++ = *src++;
}
#pragma bank_optimal_width(__code, 16)
```

the compiler knows that the instructions for the generated function would be best fetched in multiples of 16 bits, and so can select instructions accordingly.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition. This is shown in the preceding example.

Increments and Decrements

Care should be taken when using increments or decrements in expressions. The C and C++ standards do not define the order that operations should take place in every circumstance.

For example, if in the following assignment expression,

```
a[i] = i++;
```

the intention might be to put the value of `i` into array `a` at index `i`, then increment `i` ready for assignment into the next element of `a`. However, it is up to the compiler when to perform the post-increment operation to make full use of the functionality available on the target machine. In such cases, the behavior of the code may not be as the user intended.

To be safe, it is recommended that you do not use an increment or decrement operator with a variable that appears more than once in a single expression. Instead, separate out the increment or decrement and put it in the desired place.

C++ Style Comments

The compiler accepts C++ style comments, beginning with `//` and ending at the end of the line, in C programs. This is essentially compatible with standard C, except for the following case.

```
a = b
//* highly unusual */ c
:
```

which a standard C compiler processes as:

```
a = b/c;
```

and a C++ compiler and `ccts` process as:

```
a = b;
```

C++ Fractional Type Support

While in C++ mode, the `ccts` compiler supports fractional (fixed-point) arithmetic that provides a way of computing with non-integral values within the confines of the fixed-point representation. Hardware support for the 32-bit fractional arithmetic is available on the TigerSHARC processors.

This section describes:

- "Format of Fractional Literals" on page 1-234
- "Conversions Involving Fractional Values" on page 1-235
- "Fractional Arithmetic Operations" on page 1-235
- "Mixed-Mode Operations" on page 1-236

Fractional values are declared with the `fract` data type. Ensure that your program includes the `<fract>` header file. The `fract` data type is a C++ class that supports a set of standard arithmetic operators used in arithmetic expressions. Fractional values are represented as signed values in a range of `[-1.0 … +1.0)` with a binary point immediately after the sign bit. Other value ranges are obtained by scaling or shifting. In addition to the arithmetic, assignment, and shift operations, `fract` provides several type-conversion operations.

Note that this implementation does not provide for automatic scaling of fractional values.

Format of Fractional Literals

Fractional literals use the floating-point representation with an "`r`" suffix to distinguish them from floating-point literals; for example, `0.5r`. The `ccts` compiler validates fractional literal values to ensure they reside within the valid range of values.

Fractional literals are written with the "r" suffix to avoid certain precision loss. Literals without an "r" are of the type double, and are implicitly converted to fract as needed. After the conversion of a 32-bit double literal to a fract literal, the value of the latter retains only 24 bits of precision compared with the full 32 bits for a fractional literal with the "r" suffix.

Conversions Involving Fractional Values

The following notes apply to type-conversion operations:

- Conversion between a fractional value and a floating value is supported. The conversion to the floating-point type may result in some precision loss.
- Conversion between a fractional value and an integer value is not supported. The conversion is not recommended because the only common values are 0 and –1.
- Conversion between a fractional value and a long double value is supported via float and may result in some precision loss.

Fractional Arithmetic Operations

The following notes summarize information about fractional arithmetic operators supported by the compiler:

- Standard arithmetic operations on two fract items include addition, subtraction, and multiplication.
- Assignment operations include +=, -=, and *=.
- Shift operations include left and right shifts. A left shift is implemented as a logical shift and a right shift is an arithmetic shift. Shifting left by a negative amount is not recommended.
- Comparison operations are supported between two fract items.

- When arithmetic expressions contain a mixture of `fract` and `float` (or `double`) items, then the `float` (or `double`) items are normally converted to `fract` representation. For more information, see "Mixed-Mode Operations" on page 1-236.
- Multiplication of a fractional and an integer produces an integer result or a fractional result. The program context determines which type of result is generated following the conversion algorithm of C++. When the compiler does not have enough context, it generates an ambiguous operator message.

  For example,

  ```
  error: more than one operator "*" matches these operands:

  ...
  ```

  You must explicitly cast the result of the multiply operation if the error occurs.

Mixed-Mode Operations

Most operations that are supported for fractional values are supported for mixed fractional/float or fractional/double arithmetic expressions. At runtime, a floating-point value is converted to a fractional value, and the operation is completed using fractional arithmetic.

The assignment operations, such as +=, are the exception to the rule. The logic of an assignment operation is defined by the type of a variable positioned on the left side of the expression.

Floating-point operations require an explicit cast of a fractional value to the desired floating type.

GCC Compatibility Extensions

The compiler provides compatibility with the C dialect accepted by version 3.2 of the GNU C Compiler. Many of these features are available in the C99 ANSI Standard. A brief description of the extensions is included in this section. For more information, refer to the following web address:

`http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/C-Extensions.html#C%20Extensions`

The GCC compatibility extensions are only available in C dialect mode. They are not accepted in C++ dialect mode.

Statement Expressions

A statement expression is a compound statement enclosed in parentheses. A compound statement itself is enclosed in braces { }, so this construct is enclosed in parentheses-brace pairs ({ }).

The value computed by a statement expression is the value of the last statement (which should be an expression statement). The statement expression may be used where expressions of its result type may be used. But they are not allowed in constant expressions.

Statement expressions are useful in the definition of macros as they allow the declaration of variables local to the macro. In the following example,

```
#define min(a,b) ({                 \
   short __x=(a),__y=(b),__res;     \
   if (__x > __y)                   \
      __res = __y;                  \
   else                             \
      __res = __x;                  \
   __res;                           \
})

int use_min() {
   return  min(foo(), thing()) + 2;
}
```

The `foo()` and `thing()` statements get called once each because they are assigned to the variables `__x` and `__y` which are local to the statement expression that `min` expands to. The `min()` can be used freely within a larger expression because it expands to an expression.

Labels local to a statement expression can be declared with the `__label__` keyword. For example,

```
({
   __label__ exit;
   int i;
   for (i=0; p[i]; ++i) {
      int d = get(p[i]);
      if (!check(d)) goto exit;
      process(d);
   }
exit:
   tot;
})
```

Statement expressions are not supported in C++ mode.

Statement expressions are an extension to C originally implemented in the GCC compiler. Analog Devices support the extension primarily to aid porting code written for that compiler. When writing new code, consider using inline functions, which are compatible with ANSI/ISO standard C++ and C99, and are as efficient as macros when optimization is enabled.

Type Reference Support Keyword (typeof)

The `typeof( expression )` construct can be used as a name for the type of expression without actually knowing what that type is. It is useful for making source code that is interpreted more than once, such as macros or include files, more generic.

The `typeof` keyword may be used wherever a `typedef` name is permitted such as in declarations and in casts. For example,

```
#define abs(a) ({                    \
   typeof(a) __a = a;                \
   if (__a < 0) __a = - __a;         \
   __a;                              \
})
```

shows `typeof` used in conjunction with a statement expression to define a "generic" macro with a local variable declaration.

The argument to `typeof` may also be a type name. Because `typeof` itself is a type name, it may be used in another `typeof( type-name )` construct. This can be used to restructure the C-type declaration syntax.

For example,

```
#define pointer(T)   typeof(T *)
#define array(T, N)  typeof(T [N])

array (pointer (char), 4) y;
```

declares `y` to be an array of four pointers to `char`.

The `typeof` keyword is not supported in C++ mode.
The `typeof` keyword is an extension to C originally implemented in the GCC compiler. It should be used with caution because it is not compatible with other dialects of C or C++ and has not been adopted by the more recent C99 standard.

GCC Generalized Lvalues

A cast is an `lvalue` (may appear on the left-hand side of an assignment) if its operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC. It is not allowed in C++ mode.

A comma operator is an `lvalue` if its right operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC. It is a standard feature of C++.

A conditional operator is an `lvalue` if its last two operands are `lvalues` of the same type. This is an extension to C, provided for compatibility with GCC. It is a standard feature of C++.

Conditional Expressions With Missing Operands

The middle operand of a conditional operator can be left out. If the condition is nonzero (true), then the condition itself is the result of the expression. This can be used for testing and substituting a different value when a pointer is NULL. The condition is only evaluated once; therefore, repeated side effects can be avoided. For example,

```
printf("name = %s\n",  lookup(key)?:"-");
```

calls `lookup()` once, and substitutes the string "-" if it returns NULL. This is an extension to C, provided for compatibility with GCC. It is not allowed in C++ mode.

Hexadecimal Floating-Point Numbers

C99 style hexadecimal floating-point constants are accepted. They have the following syntax.

```
hexadecimal-floating-constant:
  {0x|0X} hex-significand binary-exponent-part [ floating-suffix ]
hex-significand: hex-digits [ . [ hex-digits ]]
binary-exponent-part: {p|P} [+|-] decimal-digits
floating-suffix: { f | l | F | L }
```

The hex-significand is interpreted as a hexadecimal rational number. The digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of two by which the significand is to be scaled. The floating suffix has the same meaning that it has for decimal

floating constants—a constant with no suffix is of type `double`, a constant with suffix `F` is of type `float`, and a constant with suffix `L` is of type `long double`.

Hexadecimal floating constants enable the programmer to specify the exact bit pattern required for a floating-point constant.

For example, the declaration

```
float f = 0x1p-126f;
```

causes `f` to be initialized with the value `0x800000`.

Zero-Length Arrays

Arrays may be declared with zero length. This is an anachronism supported to provide compatibility with GCC. Use variable-length array members instead.

Variable Argument Macros

The final parameter in a macro declaration may be followed by ... to indicate the parameter stands for a variable number of arguments.

For example,

```
#define trace(msg, args...)  fprintf (stderr, msg, ## args);
```

can be used with differing numbers of arguments,

```
trace("got here\n");
trace("i = %d\n", i);
trace("x = %f, y = %f\n", x, y);
```

The ## operator has a special meaning when used in a macro definition before the parameter that expands the variable number of arguments: if the parameter expands to nothing, then it removes the preceding comma.

The variable argument macro syntax comes from GCC. It is not compatible with C99 variable argument macros and is not supported in C++ mode.

Line Breaks in String Literals

String literals may span many lines. The line breaks do not need to be escaped in any way. They are replaced by the character `\n` in the generated string. This extension is not supported in C++ mode. The extension is not compatible with many dialects of C, including ANSI/ISO C89 and C99. However, it is useful in `asm` statements, which are intrinsically non-portable.

Arithmetic on Pointers to Void and Pointers to Functions

Addition and subtraction is allowed on pointers to `void` and pointers to functions. The result is as if the operands had been cast to pointers to `char`. The `sizeof()` operator returns one for `void` and function types.

Cast to Union

A type cast can be used to create a value of a union type, by casting a value of one of the union member's types.

Ranges in Case Labels

A consecutive range of values can be specified in a single case by separating the first and last values of the range with ....

For example,

```
case 200 ... 300:
```

Declarations Mixed With Code

In C mode, the compiler accepts declarations placed in the middle of code. This allows the declaration of local variables to be placed at the point where they are required. Therefore, the declaration can be combined with initialization of the variable.

For example, in the following function

```
void func(Key k) {
   Node *p = list;
   while (p && p->key != k)
      p = p->next;
   if (!p)
      return;
   Data *d = p->data;
   while (*d)
      process(*d++);
}
```

the declaration of `d` is delayed until its initial value is available, so that no variable is uninitialized at any point in the function.

Escape Character Constant

The character escape "`\e`" may be used in character and string literals and maps to the ASCII Escape code, 27.

Alignment Inquiry Keyword (__alignof__)

The `__alignof__ (type-name)` construct evaluates to the alignment required for an object of a type. The `__alignof__` *`expression`* construct can also be used to give the alignment required for an object of the *`expression`* type.

If expression is an `lvalue` (may appear on the left-hand side of an assignment), the alignment returned takes into account alignment requested by pragmas and the default variable allocation rules.

(asm) Keyword for Specifying Names in Generated Assembler

The `asm` keyword can be used to direct the compiler to use a different name for a global variable or function. (See also "#pragma linkage_name identifier" on page 1-206.)

For example,

```
int N asm("C11045");
```

tells the compiler to use the label C11045 in the assembly code it generates wherever it needs to access the source level variable `N`. By default, the compiler would use the label `_N`.

The `asm` keyword can also be used in function declarations but not function definitions. However, a definition preceded by a declaration has the desired effect. For example,

```
extern int f(int, int) asm("func");

int f(int a, int b) {
. . .
}
```

Function, Variable and Type Attribute Keyword (__attribute__)

The `__attribute__` keyword can be used to specify attributes of functions, variables and types, as in these examples:

```
void func(void) __attribute__ ((section("fred")));
int a __attribute__ ((aligned (8)));
typedef struct {int a[4];} __attribute__((aligned (4))) Q;
```

The `__attribute__` keyword is supported, and therefore code, written for GCC, can be ported. All attributes accepted by GCC on ix86 are accepted. The ones that are actually interpreted by the `ccts` compiler are described in the sections of this manual describing the corresponding pragmas. (See "Pragmas" on page 1-179.)

Unnamed struct/union fields within struct/unions

The compiler allows you to define a structure or union that contains, as fields, structures and unions without names. For example,

```
struct {
    int field1;
    union {
        int field2;
        int field3;
    };
    int field4;
} myvar;
```

This allows the user to access the members of the unnamed union as though they were members of the enclosing struct, for example, `myvar.field2`.

Preprocessor-Generated Warnings

The preprocessor directive `#warning` causes the preprocessor to generate a warning and continue preprocessing. The text that follows the `#warning` directive on the line is used as the warning message.

Preprocessor Features

The `ccts` compiler provides standard preprocessor functionality, as described in any C text. The following extensions to standard C are also supported:

```
// end of line (C++-style) comments
#warning directive
```

For more information about these extensions, refer to "Preprocessor-Generated Warnings" on page 1-245.

This section describes:

- "Predefined Preprocessor Macros"
- "Writing Macros" on page 1-249

Predefined Preprocessor Macros

The `ccts` compiler defines a number of macros to produce information about the compiler, source file, and options specified. These macros can be tested, using the `#ifdef` and related directives, to support your program's needs. Similar tailoring is done in the system header files.

Macros, such as `__DATE__`, can be useful to incorporate in text strings. The "#" operator within a macro body is useful in converting such symbols into text constructs.

Table 1-26 lists the predefined preprocessor macros.

Table 1-26. Predefined Preprocessor Macros

Preprocessor Macro	Description
__ADSPTS__	ccts always defines __ADSPTS__ as 1, indicating that a TigerSHARC program is being compiled
__ADSPTS101__	ccts defines __ADSPTS101__ as 1 when you compile with the -proc ADSP-TS101 command-line switch
__ADSPTS201__	ccts defines __ADSPTS201__ as 1 when you compile with the -proc ADSP-TS201 command-line switch
__ADSPTS202__	ccts defines __ADSPTS202__ as 1 when you compile with the -proc ADSP-TS202 command-line switch
__ADSPTS203__	ccts defines __ADSPTS203__ as 1 when you compile with the -proc ADSP-TS203 command-line switch
__ADSPTS20x__	ccts defines __ADSPTS20x__ as 1 when you compile with either the -proc ADSP-TS201, -proc ADSP-TS202, or -proc ADSP-TS203 command-line switch
__ANALOG_EXTENSIONS__	ccts defines __ANALOG_EXTENSIONS__ as 1, unless you compile with -pedantic or -pedantic-errors
__cplusplus	ccts defines __cplusplus as 1 when you compile in C++ mode
__DATE__	The preprocessor expands this macro into the current date as a string constant. The date string constant takes the form Mmm dd yyyy (ANSI standard).
__DOUBLES_ARE_FLOATS__	The macro __DOUBLES_ARE_FLOATS__ is defined to 1 when the size of the double type is the same as the single precision float type. When the compiler switch -double-size-64 is used, the macro is not defined.
__ECC__	ccts always defines __ECC__ as 1
__EDG__	ccts always defines __EDG__ as 1, which signifies that an Edison Design Group front end is being used
__EDG_VERSION__	ccts always defines __EDG_VERSION__ as an integral value representing the version of the compiler's front end

Table 1-26. Predefined Preprocessor Macros (Cont'd)

Preprocessor Macro	Description
__FILE__	The preprocessor expands this macro into the current input file name as a string constant. The string matches the name of the file specified on the ccts command line or in a preprocessor #include command (ANSI standard).
__LINE__	The preprocessor expands the __LINE__ macro into the current input line number as a decimal integer constant (ANSI standard)
__NO_BUILTIN	ccts defines __NO_BUILTIN as 1 when you compile with the -no-builtin command-line switch
__SIGNED_CHARS__	ccts defines __SIGNED_CHARS__ as 1 unless you compile with the -unsigned-char command-line switch
__STDC__	ccts always defines __STDC__ as 1
__STDC_VERSION__	ccts always defines __STDC_VERSION__ as 199409L
__TIME__	The preprocessor expands this macro into the current time as a string constant. The time string constant takes the form hh:mm:ss (ANSI standard).
__TS_BYTE_ADDRESS	ccts defines this macro if byte-addressing mode is selected using the -char-size-8 switch
__SILICON_REVISION__	The __SILICON_REVISION__ macro is defined when the -si-revision switch is specified with a value other than "none". The value it is defined to is the major revision number left-shifted by 8 logically or'd against the minor revision number.
__VERSION__	The preprocessor expands this macro into a string constant containing the current compiler version

Table 1-26. Predefined Preprocessor Macros (Cont'd)

Preprocessor Macro	Description
__VERSIONNUM__	Defines __VERSIONNUM__ as a numeric variant of __VERSION__ constructed from the version number of the compiler. Eight bits are used for each component in the version number and the most significant byte of the value represents the most significant version component. As an example, a compiler with version 7.1.0.0 defines __VERSIONNUM__ as 0x07000100 and 7.1.1.10 would define __VERSIONNUM__ to be 0x0701010A.
__WORKAROUNDS_ENABLED	Defines this macro to be 1 if any hardware workarounds are implemented by the compiler. This macro is set if the -si-revision switch has a value other than "none" or if any specific workaround is selected by means of the -workaround switch.

Writing Macros

A macro is a name standing for a block of text that the preprocessor substitutes for. Use the #define preprocessor command to create a macro definition. When the macro definition has arguments, the block of text the preprocessor substitutes can vary with each new set of arguments.

Compound Statements as Macros

When writing macros, it can be useful to define a macro that expands into a compound statement. You can define such a macro so it can be invoked in the same way you would call a function, making your source code easier to read and maintain.

The following two code segments define two versions of the macro SKIP_SPACES:

```
/* SKIP_SPACES, regular macro */
#define SKIP_SPACES ((p), limit)    { \
   char *lim = (limit); \
      while ((p) != lim)            { \
         if (*(p)++ != ' ')         { \
            (p)– –; \
```

```
            break; \
         } \
      } \
}

/* SKIP_SPACES, enclosed macro */
#define SKIP_SPACES (p, limit)   \
   do { \
      char *lim = (limit);       \
      while ((p) != lim)       { \
         if (*(p)++ != ' ')    { \
            (p)– –;  \
            break;  \
         }  \
      }  \
   } while (0)
```

Enclosing the first definition within the `do {…} while (0)` pair changes the macro from expanding to a compound statement to expanding to a single statement. With the macro expanding to a compound statement, you would sometimes need to omit the semicolon after the macro call in order to have a legal program. This leads to a need to remember whether a function or macro is being invoked for each call and whether the macro needs a trailing semicolon or not.

With the `do {…} while (0)` construct, you can pretend that the macro is a function and always put the semicolon after it. For example,

```
/* SKIP_SPACES, enclosed macro, ends without ‘;’ */
if (*p != 0)
   SKIP_SPACES (p, lim);
else...
```

This expands to:

```
if (*p != 0)
   do {
      ...
   } while (0);   /* semicolon from SKIP_SPACES (...);  */
else...
```

Without the `do {...} while (0)` construct, the expansion would be:

```
if (*p != 0)
   {
      ...
   }
;   /* semicolon from SKIP_SPACES (...); */
else...
```

The above code is not legal C or C++ syntax.

C/C++ Run-Time Model and Environment

This section provides a full description of the TigerSHARC run-time model and run-time environment. The run-time model, which applies to compiler-generated code, includes descriptions of the layout of the stack, data access, and call/entry sequence. The C/C++ run-time environment includes the conventions that C/C++ routines must follow to run on TigerSHARC processors. Assembly routines linked to C/C++ routines must follow these conventions.

ADI recommends that assembly programmers maintain stack conventions.

This section provides:

- "Stack Frame Overview" on page 1-253
- "Stack Frame Description" on page 1-255
- "Support for argv/argc" on page 1-264
- "Using Multiple Heaps" on page 1-266
- "Miscellaneous Information" on page 1-273
- "Register Classification" on page 1-273

Stack Frame Overview

The stack frame (or activation record) provides for the following activities:

- Space for local variables for the current procedure. For the compiler, this includes temporary storage as well as that required for explicitly declared user automatic variables.
- Place to save linkage information, such as return addresses, location information for the previous caller's stack frame, and to allow this procedure to return to its caller.
- Space to save information that must be preserved and restored.
- Arguments passed to the current procedure.

In addition, if this is not a leaf procedure (if it is going to call other procedures), its stack frame also contains outgoing linkage and parameter space:

- Space for the arguments to the called procedure.
- Space for the callee to save basic linkage information.

Figure 1-3 provides a general overview of the stack. Note that the stack grows downward on the page. Because the stacks grow towards smaller addresses, higher addresses are found in the upwards direction. Each row is a quad word group; the higher addresses are on the left since this is a little-endian machine.

There are two stacks and thus two current stack frames at any point on the TigerSHARC processors. One stack resides in each of the data memory banks. One stack is addressed from the `j iALU` registers, the other stack is addressed from the `k` register. For more information, refer to "Stack Frame Description" on page 1-255.

Each stack is controlled by a pair of pointers—a Stack Pointer (SP), which identifies the boundary of the in-use portion of the stack space, and a

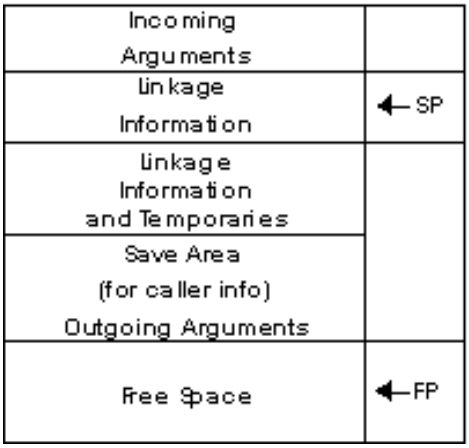

Figure 1-3. ADSP-TS101 Processor Stack

Frame Pointer (FP), which provides stable addressing to the current frame.

The second stack is not yet used for local variables.

The following design attributes have been included in the TigerSHARC processor's run-time model for additional efficiency:

- The frame pointers are offset by `-0x40` from the actual base of the frame. This provides greater addressing range for larger negative offsets (for local variables) than positive ones (for arguments). Since all FP-based references are relative. (For example, with an offset), there is no run-time cost and little extra complexity. This offset is constant throughout the system; it does not change for particular routines. However, this may cause the Frame Pointer address to actually be off the end of the stack.
- Optionally, a copy of the `j` frame pointer may be kept in register `k24`. This increases performance by allowing simple `jFP`-relative memory references to occur in the same instruction line as other

[address] calculations in the jALU. This copy is made at the discretion of the particular procedure. A similar copy of kSP into j is also possible.

- A routine that does not wish to use the k stack need not do so. If it does not use the k stack, it can dispense with all operations, including linkage related to the k stack and k stack pointers.

At present, a k frame must always be allocated in order for the debugger to be able to walk back up the stack. At some future point, it is expected that the debug information will be enriched to include information on whether or not a k frame is present.

- Space is allocated in the caller's frame for at least four "argument words". (See in "Stack Frame Description".) For routines with few arguments, this space is available for use as temporary storage. That may be of use for small, leaf routines, which may be able to avoid having to create a stack frame.

Stack Frame Description

This section describes the stack as shown in Figure 1-3 on page 1-254.

Incoming Arguments

The memory area for incoming arguments begins at the effective jSP value + 0x8 and after the frame pointer has been modified the arguments can be accessed from j26 + 0x48. Argument words are mapped by ascending addresses, so the second argument word is mapped at j26+0x49. Note that the first four words arrive in registers, although space is allocated in memory as well.

Linkage Information

The return address is saved in the CJMP register by the CALL instruction. This is saved at jSP+0 (j27+0) on entry. It is retrieved from effective jFP + 0 (j26+64) at exit.

Local Variables and Temporaries

Space for a register save area and local variables/temporaries is allocated on both stacks by the procedure prologue. The save area begins after the outgoing argument space, typically at `jSP+12` and `kSP+8`. This can also be addressed from `jFP` and `kFP`. Local variable/temp storage begins after the save area. It is addressed at effective `jFP`-offset or effective `kSP`-offset. Since the actual `FP` registers are offset by a constant `0x40`, the actual offsets might be positive for some cases. This should cause no problems.

Outgoing Arguments

Space for outgoing arguments is allocated on the stack at `jSP+8 (j27+8)`. Four words are always available as part of the basic stack frame. If more are needed, they can either be allocated as part of the prologue or by local extension of the stack frame for the purpose of a particular call.

Outgoing Linkage

Four words, at `jSP+4 (j27+4)` and `kSP+4 (k27+4)`, are available immediately to the callee for saving linkage information. These words are designated for holding the quad registers containing the `j` and `k` stack pointer registers. Four words, at `jSP+0 (j27+0)` and `kSP+0 (k27+0)`, are also available at any time, and may be used by the callee.

Free Space

Space below `jSP` and `kSP` is considered free and unprotected. It is available for use (in growing the stack) at any time, synchronously or asynchronously (the latter for interrupt handling). This space should never be used.

General System-Wide Specifications

The C/C++ run-time model assumes that the stack is placed in internal memory. As described in "Stack Frame Overview" on page 1-253, there are many instances in compiler-generated code and in the run-time libraries where the stack is accessed and there would be a significant performance overhead if the stack were to be located in external memory. As the stack is assumed to be located in internal memory, the compiler and run-time libraries make use of loads directly into sequencer registers which would be illegal were the stack to be placed in external memory. When assembling code generated by the compiler, the driver suppresses diagnostic reports from the assembler regarding loads into sequencer registers.

Some general specifications that apply to the stacks are:

- The stacks grow down in memory from higher to lower addresses.
- The current frames' "base" is addressed by the `FP` register (the value in `FP` plus `0x40`).
- The first free quad-word in each stack is addressed by the `SP` register. Locations at that point and beyond are vulnerable and must not be used. These locations may be clobbered by asynchronous activities, such as interrupt service routines. Alternatively, locations at SP and beyond are always available if additional space is needed, but SP must be moved to guard the space. Therefore, the first "used" (protected) word is at `SP+4`.

Data can be pushed onto the stack by executing an instruction like `[jSP += -4] = reg;` for either `j` or `k`.

- The address of each `FP` must always be four-word aligned; that is, the low-order two address bits are always zero.

- Likewise, the SP registers should also be kept four-word aligned at all times. This allows interrupt routines to save registers without having to first verify stack alignment.
- The return address of the caller is stored at offset zero from the address carried by the current effective jFP.
- The linkage back to the previous stack frame is stored at offset +6 and +7 from each current effective FP.

At a procedure call, the following must be true:

- Each SP must be four-word aligned (as it forms the basis for the new frame's FP).
- There must be eight words available starting at jSP+4 and four words available starting at kSP+4. The first four words on each stack are used for storing linkage information, and the remaining four (and perhaps others) on the j stack hold arguments. There is always space allocated for at least four arguments, even if the current procedure does not have that many. That way, the callee can always store the argument registers. Small routines are free to use unneeded argument slots for local storage.
- The standard calling convention further specifies that the previous frame's linkage information is already stored in the standard slots by the time control reaches the new procedure. However, the callee is responsible for restoring this information prior to exit.

  The saving of the frame linkage can be performed as part of the instruction line containing the procedure call. In some instances, this information remains constant for several outgoing calls, in which case it need not be repeatedly stored.

Argument Passage

A contiguous block of memory, near the end of the current frame, is provided to hold all arguments. Argument handling begins by conceptually "mapping" all the arguments into this area. The arguments are laid out into ascending addresses. They must obey alignment constraints based on their size. An argument might occupy more than one word; there might also be a vacant word, or "hole", in the argument area to maintain alignment. This is the definition of "argument words".

Data corresponding to the first four words of the argument area are normally passed in registers, rather than on the stack, which increases efficiency. For a common situation where each argument occupies a single word, the first four arguments can be passed in registers. There is also space allocated on the stack so that these register values can be stored back in the prologue in their canonical places.

The choice of register file used for argument passage is based on the argument's declared type. Although two register files are available, only four argument words are passed. The corresponding word in the other register file is unused. (See Table 1-27 on page 1-260.)

Rules governing argument words are:

- Pointers and integral types of one word are passed in the j registers.
- Floating types, integral types of more than one word, and structure (or unions) that fit, are passed in the x registers. This use of registers to pass structures or unions operates independently of the type of the structure or union fields. (A structure composed entirely of pointers is still passed in x.) The alignment constraints are still observed.
- If an argument requires sufficient space that causes it to span over the end of the register argument area (typically, if it requires more than two words), it and all succeeding arguments are passed in memory in the usual place rather than in the registers.

- If the prototype specifies `varargs ("…")`, then the argument immediately preceding the ellipsis and all succeeding arguments are passed in memory. Note that `varargs` routines MUST have prototypes in order to function correctly.
- If there is no prototype, the actual types of the arguments are used to determine how they are passed.
- Large structures may be passed by value. The structures are copied into the argument area, just as with other arguments.

Table 1-27. Register–Argument Word Correspondence

Argument Word	Register Used	
	if pointer or int	if float or double word
Arg word 1	j4	xR4
Arg word 2	j5	xR5
Arg word 3	j6	xR6
Arg word 4	j7	xR7

Return Values

Return values always use registers. The type of the value determines, as with arguments, whether to use the `j` or `x` register file (`j8`, or `xR8` and `xR9`).

Two `x` return registers are identified to accommodate double-word result values.

If the return value is larger than two words, then the caller must allocate space and pass the address in as a "hidden argument". The `j` return register, `j8`, is used for this purpose. The procedure must also return the [same] pointer value in that register on return. This is an optimization to allow more efficient referencing of the returned value.

Procedure Call and Return

The following steps describe how to manage procedure calls and returns.

To Call a Procedure:

1. Expand the current stack frame, if necessary, to provide space for the outgoing arguments and linkage information.

   The number of arguments must be rounded up to a multiple of four in order to maintain proper alignment.

   It is recommended that you establish and retain a basic call area during prologue. This saves you from creating and removing the area at each procedure call. If the number of arguments is large or a nested call occurs you must do a full expand-remove cycle.

2. Evaluate the arguments and set them up in the argument area and/or registers.

   If another function must be called as part of computing an argument, then the stack can be extended (temporarily) a second time. If you extend the stack a second time, it's probably best to store even the register arguments of the outer procedure into the stack (for safekeeping), then load them just prior to the actual call.

3. Call the procedure, saving the current Frame Pointers in the appointed slots.

4. On return, remove the frame expansion if desired.

On Entry:

1. Set the new Frame Pointers (in both `iALU`s) to the value of the current stack pointer, less the `0x40` offset. This can be done simultaneously in both `iALU`s.

2. Set up a new frame and continue saving context: store the cjmp (return) register on the `j` stack at offset +0 from the current (old) stack pointer. By using post-modify addressing on the store

instruction, the stack pointer may simultaneously be moved down to create a new frame.

Registers are saved on the `k` stack as needed while the frame is created.

After this step, the new frame is officially in place.

3. If debug is specified (the `-g` switch), arguments arriving in registers should be stored back into memory so the debugger can find them.

This restriction may be lifted in future releases when the debug tables are more expressive.

4. If desired, transfer a copy of the new `jFP` register to `k`.
5. Continue saving registers and then executing the procedure.

A leaf procedure that does not require much stack space might choose to omit steps (1) and (2), operating without its own stack frame. A small amount of local storage is available in any unused argument slots since there are always at least four slots as well as the two quad words reserved for saving linkage information.

After this step, the new frame is officially in place.

6. Continue saving registers, and then executing the procedure.

To Return from a Procedure:

1. If a call was made to another procedure, then `CJMP` must be restored.

   For best performance, restore `CJMP` as soon as possible after the last procedure call or computed jump, (such as a switch). If the return jump is predicted, stalls can be avoided provided that `CJMP` is reloaded four cycles plus twelve (12) words prior to the return.

2. Restore miscellaneous saved registers.
3. Place the return value in the correct register (if not there already).

4. Restore the Stack Pointers for the previous frame. This can be accomplished, for each stack, with a single quad load.
5. Return to the caller.

Code Sequences

The following example shows the normal code sequences used for the various actions involved in calling a procedure and returning from a procedure.

```
// Extend stack frame if necessary (more than 4 arguments)
jsp = jsp - nn; ksp = ksp - mm;;      // must be multiple of 4
// Evaluate arguments. Store into memory, or into arg registers.
...
// Now do the call.
call subr; q[jSP+4]=j27:24; q[kSP+4]=k27:24;;
```

The fourth slot might be available for some computation or it may be used by the `IMEX` forming the full address for the call.

```
// if the stack was extended, cut it back.
jSP = jSP + nn; kSP = kSP + mm;;   // must be multiple of 4
```

Prologue:

```
jFP = jSP -0x40; kFP = kSP -0x40;;
[jSP += -nnj]=CJMP; q[kSP += -nnk]=<some regs>;;
         // At this point, the new frame is intact.
q[jSP+n] = <some quad>; kjFPcopy = jFP;
```

Epilogue:

```
// Restore CJMP after last outgoing call
CJMP = [jFP+0x40];
```

More code:

```
// Nearing end, restore other saved registers.
...
// Exit: Restore stack linkage and return.
jump CJMP; j27:24 = q[jFP+0x44]; k27:24=q[kFP+0x44];;
```

There is a four-cycle wait before either frame pointer can be used again. If the return jump is not predicted correctly, the four-cycle wait covers the delay. Otherwise, callers might want to refrain from immediately accessing the frame pointers on return.

You should not reset the stack pointers prior to return. That would create a time window in which the stack has been reset to a frame not corresponding to the current procedure.

Support for argv/argc

By default, the facility to specify arguments that get passed to your `main()` (`argv/argc`) at run-time is enabled. However, to correctly set up `argc` and `argv` requires additional configuration by the user. Modify your application in the following ways:

1. Define your command-line arguments in C by defining a variable called "`__argv_string`". When linked, your new definition overrides the default zero definition otherwise found in the C run-time library. For example,

   ```
   const char __argv_string[] = "-in x.gif -out y.jpeg";
   ```

2. To use command-line arguments as part of Profile-Guided Optimizations (PGO), it is necessary to define `__argv_string` within a memory section called `SEG_ARGV`. Therefore, define a memory section called `seg_argv` in your `.LDF` file and include the definition of `___argv_string` in it if using PGO. The default `.LDF` files do this for you if macro `IDDE_ARGS` is defined at link time.

Allocation of Memory for Stacks and Heaps in LDFs

In previous releases of VisualDSP++, the default stacks and heaps were allocated separate memory sections in the LDFs. In VisualDSP++ 4.5, the allocation of memory for stacks and heaps is performed by the linker at link-time, resulting in more efficient memory use. The memory allocation method is as follows:

- An area of memory in one of the default memory areas (for example, `M1Data`) is reserved for stacks and heaps, using the `RESERVE()` command.
- Memory is allocated to data that must be placed in this section (for example, global variables and static variables).
- The `RESERVE_EXPAND()` command is use to claim any unused space in the default memory area and allocate it to the stack and heap. The ratio of memory allocated to the stack and heap can be adjusted, if necessary.

Example of Heap/Stack Memory Allocation

Listing 1-1 on page 1-265 shows how the `RESERVE_EXPAND()` command can be used to allocate memory for a heap and a stack in the LDF.

Listing 1-1. Heap/Stack Memory Allocation in LDFs

```
data1_cont
{
  INPUT_SECTIONS($OBJECTS(MEM_ARGV) $LIBRARIES(MEM_ARGV))
  // Allocate stacks for the application. Note that stacks
  // grow downward, and must be quad-word aligned. This
  // means that the location just after the highest word of
  // the stack is quad-word aligned (evenly divisible by
  // 4). There are two labels for each stack: "*_base" is
  // the location just ABOVE the top of the stack, and
  // "*_limit" is the lowest word that is part of the
  // stack. Each stack occupies all of its own
```

```
    RESERVE_EXPAND(heaps_and_stack, heaps_and_stack_length)
    ldf_jstack_end = heaps_and_stack;
    ldf_jstack_base = (ldf_jstack_end +
      (((heaps_and_stack_length * 14K) / 16K) - 4))
       & 0xfffffffc;
    ldf_jstack_limit = ldf_jstack_base - ldf_jstack_end;
    ldf_defheap_base = ldf_jstack_base + 4;
    ldf_defheap_end = (ldf_defheap_base +
      (((heaps_and_stack_length * 2K) / 16K)  - 4))
       & 0xfffffffc;
    ldf_defheap_size = ldf_defheap_end - ldf_defheap_base;
  } >M1Data
```

The following list provides required symbols used by the run-time libraries to create and manage the stack and heap. These symbols must be defined in the LDF

- `ldf_jstack_base`
- `ldf_kstack_base`
- `ldf_defheap_base`
- `ldf_defheap_size`
- `ldf_altheap_base`
- `ldf_altheap_size`

Using Multiple Heaps

The TigerSHARC C/C++ run-time library supports the standard heap management functions `calloc`, `free`, `malloc`, and `realloc`. By default, these functions access the default heap, which is defined in the standard Linker Description File and the run-time header.

User-written code can define any number of additional heaps. These additional heaps can be accessed either by the standard `calloc`, `free`, `malloc` and `realloc` functions, or via the extension routines `heap_calloc`, `heap_free`, `heap_malloc` and `heap_realloc`.

Each heap must be declared in the heap table in the `.LDF` file and the `TS\lib\src\crt_src\ts_hdr.asm` file must declare memory and section placement for the heaps. Refer to the example below on how to modify the header object and `.LDF` file. The default `ts_hdr.asm` file declares one default heap. To use a custom `ts_hdr.asm`, assemble and use it to replace the default `ts_hdr_TS101.doj` that is specified in your copy of the `.LDF` file. The calculation for a heap's size and length occur in the project's linker description file. When linking, the linker handles substitution of values to resolve the heap's definition (the `.VAR` directive in the `ts_hdr.asm` file).

Heap Identifiers

The primary heap ID is the index of the descriptor for that heap in `ts_hdr.asm`. The default heap, `seg_heap`, is always 0 and the primary heap IDs of user-defined heaps are 1, 2, 3 and so on.

Example:
The following `my.ldf` extract contains the default heap and alternate heap entries.

```
// The default heap occupies its own memory block defheapseg
{
   ldf_defheap_base = .;
   ldf_defheap_size = MEMORY_SIZEOF(M1Heap);
} >M1Heap
altheapseg
{
   ldf_altheap_base=.;
   ldf_altheap_size = MEMORY_SIZEOF(SDRAM);
} >SDRAM
```

Note that `ldf_defheap_base` and `ldf_defheap_size` are set up for use in the heap table setup.

The following `ts_hdr.asm` is an extract which contains the default heap entry and one additional alternate heap.

```
// Create the heap descriptor table and describe the default
// heap, which is the first entry in the heap descriptor
// table. The ts_exit.asm file declares a label for the end of
// this table.

.SECTION heaptab;
.GLOBAL ___heaptab_start;
___heaptab_start:
.VAR = ldf_defheap_base;  // Start of default heap
.VAR = ldf_defheap_size;  // Size of default heap,
                          // unit==sizeof(char)
.VAR = 0;                 // ID of default heap - must be 0
.VAR = ldf_altheap_base;  // Start of alt heap
 VAR = ldf_altheap_size;  // Size of alt heap, unit==sizeof(char)
.VAR = 1;             // Increment ID of alt heap must not be 0
```

Initializing the Heap

The default heap is initialized by the run-time library. Additional heaps can be initialized by a call to `heap_init` which takes the heap ID as its parameter which would normally be done at the beginning of `main`.

Using Alternate Heaps with the Standard Interface

Alternate heaps can be accessed by the standard functions `calloc`, `free`, `malloc` and `realloc`. The run-time library keeps track of the current heap, which initially is the default heap. The current heap can be changed any number of times at run time by calling `heap_switch` with the heap ID as a parameter. The standard functions `calloc` and `malloc` always allocate a new object from the current `heap`. If `realloc` is called with a null pointer, it also allocates a new object from the current `heap`.

Previously allocated objects can be deallocated with `free` or `realloc`, or resized with `realloc`, even if the current heap is now different from when the object was originally allocated. When a previously allocated object is resized with `realloc`, the returned object is always located in the same heap as the original object.

Using the Alternate Heap Interface

The C run-time library provides the alternate heap interface functions `heap_free`, `heap_calloc`, `heap_malloc`, and `heap_realloc`. These routines work exactly the same as the corresponding standard functions without the "`heap_`" prefix, except that they take an additional argument that specifies the heap ID. These functions are completely independent of the current heap setting.

Objects allocated with the alternate heap interface functions can be freed with either the `free` or `heap_free` (or `realloc` or `heap_realloc`) functions. The `heap_free` function is a little faster than `free` since it does not have to search for the proper heap. However, it is essential that the `heap_free` or `heap_realloc` functions be called with the same heap ID that was used to allocate the object being freed. If it is called with the wrong heap ID, the object would not be freed or reallocated.

The `heap_switch()` interface is not available when using the multi-threaded run-time support. However, all of the other alternate heap interface functions are available.

The actual entry point names for the alternate heap interface routines have an initial underscore, that is, they are `_heap_switch`, `_heap_calloc`, `_heap_free`, `_heap_malloc`, and `_heap_realloc`. The `stdlib.h` standard header file defines macro equivalents without the leading underscores.

Example C Program

```
// Example program using the standard heap interface along
// with an additional heap as specified in the previous example
```

```
#include <stdlib.h>
#include <stdio.h>

main()
{
  int *w, * x, *y, *z;
  heap_init(1);          // Initailiase the alternate heap
  w = malloc(1000);      // Get 1K words of default heap space
  heap_switch(1);        // Set the current heap to the alternate
  heap x = malloc(1000); // Get 1K words of alternate heap space
                        // in case it is referred to elsewhere
  y = heap_malloc(1, 1000); // By specifying the alternate heap
                       // allocate 1K words on the alternate heap
  z = malloc(1000);    // Allocate 1K words on the current
                       // (default) heap
}
```

C++ Run-time Support for the Alternate Heap Interface

The C++ run-time library provides support for allocation and release of memory from an alternative heap via the new and delete operators.

Heaps should be initialized with the C run-time functions as described. These heaps can then be used via the new and delete mechanism by simply passing the heap ID to the new operator. There is no need to pass the heap ID to the delete operator as the information is not required when the memory is released.

The routines are used as in the example below.

```
#include <heapnew>

char *alloc_string(int size, int heapID)
{
     char *retVal = new(heapID) char[size];
     return retVal;
}

void free_string(char *aString)
{
```

```
        delete aString;
    }
```

Using the Heap_Install Interface

The `heap_install()` function sets up a memory heap (`base`) with a size specified by `length` at runtime. The dynamic heap is identified by the `userid` identifier.

```
#include <stdlib.h>
int heap_install(void *base, size_t length, int userid);
```

On successful initialization, `heap_install()` returns the heap index allocated for the newly installed heap. If the operation is unsuccessful, then `heap_install()` returns -1.

Once the dynamic heap is initialized, heap space can be claimed using the `heap_malloc()` run-time library routine and associated heap management run-time library routines.

Error Conditions

The `heap_install()` function returns -1 if initialization was unsuccessful. This may be because there is not enough space available in the `__heaps` table, or if a heap with the specified userid already exists.

Examples

<< Linker Description File >>

```
MEMORY
    {
        ..
        HOST { TYPE(RAM) START(0x80000000) END(0x8FFFFFFF) WIDTH(32) }}
        ..
    }

PROCESSOR p0
    {
        ..
```

```
SECTIONS
{
   ..

   my_own_heapseg
   {
      _ldf_my_own_heap_base = .;
      _ldf_my_own_heap_size = MEMORY_SIZEOF(HOST);
      }>HOST
      ..
   }
}
```

<< C Source File >>

```
#include <stdlib.h>
#include <stdio.h>

extern int ldf_my_own_heap_base;

#define ADDR &ldf_my_own_heap_base

int main()
{
   int i;
   int id;
   int *x;

   id /* 0 */ = heap_install((void *)ADDR, 1000, 6);

   x = heap_malloc(6, 90);

   if (x) {
      for (i = 0; i < 90; i++)
          x[i] = i;
      heap_free(6, x);
   }

   return 0;
}
```

Miscellaneous Information

This section contains a number of miscellaneous aspects of the design that may be helpful in understanding stack functionality.

- Procedures without prototypes can be called successfully, provided the argument types correspond properly. Since pointers and `its` are handled in the same way, some common interface errors are tolerated. Mixing up `float` and `integer` arguments generally results in incorrect behavior.
- There is no special interface for calling system library functions. They use the standard calling convention.
- Procedures that use the `stdargs (varargs)` mechanism need to have prototypes.
- The `k` stack has a slightly irregular existence at present. There must always be a frame, but it cannot be used for local variables. These restrictions are both based on limitations in the debug information.

Register Classification

This section describes all of the TigerSHARC processor registers. Registers are listed in order of preferred allocation by the compiler. Any registers not mentioned are not preserved and should be considered “scratch.”

Callee Preserved Registers (“Preserved”)

Registers `j16` through `j25`, `k16` through `k25`, `x24` through `x31`, `y24` through `y31` are “preserved” (or dedicated). A subroutine that uses any of these registers must save (preserve) it and restore it. Any registers not mentioned are not preserved and should be considered “scratch.”

Caller Save Registers (“Scratch”)

All registers not preserved or dedicated are *scratch*. A subroutine may use a scratch register without having to save it. Any registers not mentioned are not preserved and should be considered scratch.

ADSP-TS101 and ADSP-TS20x Processor Registers

The following is a list of registers for the ADSP-TS101 processors and ADSP-TS20x processors. The registers are listed in order of preferred allocation by the compiler.

- Table 1-28 on page 1-275 — IALU (j & k) General Registers
- Table 1-29 on page 1-276 — Compute Block General Registers
- Table 1-30 on page 1-277 — IALU (j & k) Special Registers
- Table 1-31 on page 1-277 — Compute Block X MAC Registers
- Table 1-32 on page 1-277 — Compute Block Y MAC Registers
- Table 1-33 on page 1-277 — Compute Block ALU Summation Registers
- Table 1-34 on page 1-278 — Loop Counters
- Table 1-35 on page 1-278 — Data Alignment Registers
- Table 1-36 on page 1-278 — Bit FIFO Temporaries Registers
- Table 1-37 on page 1-278 — Static Condition Flags Registers
- Table 1-38 on page 1-278 — Return Address Registers
- Table 1-39 on page 1-278 — Enhanced Communications Registers (ADSP-TS101 processors only)
- Table 1-40 on page 1-285 — Enhanced Communications Registers (ADSP-TS201 processors only)

Table 1-28. iALU (j and k) General Registers

j31:	j30:	j29:	j28:	k31:	k30:	k29:	k28:
zero[1]	scratch	scratch	scratch	zero1	scratch	scratch	scratch

j27:	j26:	j25:	j24:	k27:	k26:	k2:	k2:
Dedicated jSP	Dedicated jFP	RES (or GOT)	RES	Dedicated kSP	Dedi-cated kFP	RES (or GOT)	RES kjFP Copy

j23:	j22:	j21:	j20:	k23:	k22:	k21:	k20:
RES	RES	RES	RES	RES	RES	RES	RES

j19:	j18:	j17:	j16:	k19:	k18:	k17:	k16:
RES	RES	RES	RES	RES	RES	RES	RES

j15:	j14:	j13:	j12:	k15:	k14:	k13:	k12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

j11:	j10:	j9:	j8:	k11:	k10:	k9:	k8:
scratch	scratch	scratch	scratch RTN0[2]	scratch	scratch	scratch	scratch

j7:	j6:	j5:	j4:	k7:	k6:	k5:	k4:
scratch ARG4	scratch ARG3	scratch ARG2	scratch ARG1	scratch	scratch	scratch	scratch

Table 1-28. iALU (j and k) General Registers (Cont'd)

j3:	j2:	j1:	j0:	k3:	k2:	k1:	k0:
scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)

1 j/k 31 are zero registers when used for addressing, or as operands to iALU operations; but when used in a store or load (or ureg transfer), they become the j/k status register.
2 j8 is also used for passing the address of a hidden argument.

Table 1-29. Compute Block (x & y) General Registers

x31:	x30:	x29:	x28:	y31:	y30:	y29:	y28:
RES	RES	RES	RES	RES	RES	RES	RES

x27:	x26:	x25:	x24:	y27:	y26:	y25:	y24:
RES	RES	RES	RES	RES	RES	RES	RES

x23:	x22:	x21:	x20:	y23:	y22:	y21:	y20:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch
x19:	x18:	x17:	x16:	y19:	y18:	y17:	y16:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

x15:	x14:	x13:	x12:	y15:	y14:	y13:	y12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

x11:	x10:	x9:	x8:	y11:	y10:	y9:	y8:
scratch	scratch	scratch RTN 1	scratch RTN 0	scratch	scratch	scratch	scratch

x7:	x6:	x5:	x4:	y7:	y6:	y5:	y4:

Table 1-29. Compute Block (x & y) General Registers (Cont'd)

scratch ARG4	scratch ARG3	scratch ARG2	scratch ARG1	scratch ARGn	scratch ARGn	scratch ARGn	scratch ARGn

x3:	x2:	x1:	x0:	y3:	y2:	y1:	y0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 1-30. iALU (j & k) Special Registers: Circular Buffering – B (base) and L (length)

jB3:	jB2:	jB1:	jB0:	kB3:	kB2:	kB1:	kB0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

jL3:	jL2:	jL1:	jL0:	kL3:	kL2:	kL1:	kL0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 1-31. Compute Block X MAC Registers

xMR4:		xMR3:	xMR2:	xMR1:	xMR0:
scratch		scratch	scratch	scratch	scratch

Table 1-32. Compute Block Y MAC Registers

yMR4:		yMR3:	yMR2:	yMR1:	yMR0:
scratch		scratch	scratch	scratch	scratch

Table 1-33. Compute Block ALU Summation Registers

xPR1:	xPR0:		yPR1:	yPR0:
scratch	scratch		scratch	scratch

Table 1-34. Loop Counters

LC1:		LC0:
scratch		scratch

Table 1-35. Data Alignment Registers

xDAB:		yDAB:
scratch		scratch

Table 1-36. Bit FIFO Temporaries Registers

xBFOTMP:		yBFOTMP:
scratch		scratch

Table 1-37. Static Condition Flags Registers

iSF0:	iSF1:		xSF0:	xSF1:		ySF0:	ySF1:
scratch	scratch		scratch	scratch		scratch	scratch

Table 1-38. Return Address Registers

CJMP:		RETI:
scratch – for caller reserved – by callee		Not Available

Table 1-39. Enhanced Communications Registers (ADSP-TS101 Processors only)[1]

xtr15:	xtr14:	xtr13:	xtr12:	ytr15:	ytr14:	ytr13:	ytr12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr11:	xtr10:	xtr9:	xtr8:	ytr11:	ytr10:	ytr9:	ytr8:

Table 1-39. Enhanced Communications Registers (ADSP-TS101 Processors only)[1] (Cont'd)

scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr7:	xtr6:	xtr5:	xtr4:	ytr7:	ytr6:	ytr5:	ytr4:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr3:	xtr2:	xtr1:	xtr0:	ytr3:	ytr2:	ytr1:	ytr0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xthr1:	xthr0:	ythr1:	ythr0:
scratch	scratch	scratch	scratch

1 The registers are listed in order of preferred allocation by the compiler.

C/C++ and Assembly Language Interface

This section describes how to call assembly language subroutines from within C or C++ programs and C or C++ functions from within assembly language programs.

Before attempting to perform either of these calls, be sure to familiarize yourself with the information about the C/C++ run-time model (including details about the stack, data types, and how arguments are handled) contained in "C/C++ Run-Time Model and Environment" on page 1-252.

This section describes:

- "Calling Assembly Subroutines From C/C++ Programs"
- "Calling C/C++ Functions From Assembly Programs" on page 1-283
- "Using Mixed C/C++ and Assembly Naming Conventions" on page 1-284
- "C++ Programming Examples" on page 1-286

Calling Assembly Subroutines From C/C++ Programs

Before calling an assembly language subroutine from a C/C++ program, create a prototype to define the arguments for the assembly language subroutine and the interface from the C/C++ program to the assembly language subroutine. Even though it is legal to use a function without a prototype in C/C++, prototypes are a strongly-recommended practice for good software engineering. When the prototype is omitted, the compiler cannot perform argument-type checking and assumes that the return value is of type integer and uses K&R promotion rules instead of ANSI promotion rules.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. Scratch registers can be used within the assembly language program without worrying about their previous contents. If more room is needed (or an existing code is used) and you wish to use the preserved registers, you *must save* their contents and then *restore* those contents before returning.

Do *not* use the dedicated or stack registers for other than their intended purpose; the compiler, libraries, debugger, and interrupt routines depend on having a stack available as defined by those registers.

The compiler also assumes the machine state does not change during execution of the assembly language subroutine.

The compiler prefaces the name of any external entry point with an underscore. Therefore, declare your assembly language subroutine's name with a leading underscore. If you intend to using the function from assembly programs as well, you might want your function's name to be just as you write it. Then you also need to tell the C/C++ compiler that it is an `asm` function, by placing `'extern "asm" {}'` around the prototype.

The C/C++ runtime determines that all function parameters are passed on the stack. A good way to observe and understand how arguments are passed is to write a dummy function in C or C++ and compile it using the `-save-temps` command-line switch. (See on page 1-59.) The resulting compiler generated assembly file (`.s`) can then be viewed.

The following example includes the global volatile variable assignments to indicate where the arguments can be found upon entry to `asmfunc`.

```
// Sample file for exploring compiler interface...
// global variables ... assign arguments there just so
// we can track which registers were used
// (type of each variable corresponds to one of arguments)

   int global_a;
   float global_b;
   int *  global_p;
```

```
// the function itself

   int asmfunc(int a, float b, int * p, int d, int e) {
// do some assignments so .s file shows where args are
   global_a = a;
   global_b = b;
   global_p = p;
         //value gets loaded into the return register
   return 12345;
  }
```

When compiled with the `-S` and `-O` options set, this produces the following code.

A TigerSHARC processor passes up to four arguments in registers. (The optimizer has gratuitously reversed the order of the assignments.)

```
// PROCEDURE: _asmfunc
   .global _asmfunc;
_asmfunc:
   J8 = j31 + 12345;;
   [j31 + _global_p] = J6;;
   [j31 + _global_b] = XR5;;
   cjmp (NP) (ABS); [j31 + _global_a] = J4;;
```

If the arguments are on the stack, they are addressed by an offset from the stack pointer or frame pointer. For a simple function, this may be all the information you need. Do the computation, set up a return value, and return to the caller.

For a more complicated function, you might find it useful to follow the general run-time model, and use the run-time stack for local storage, and so on. A simple C program, passed through the compiler, provides a good template to build on.

Calling C/C++ Functions From Assembly Programs

You may want to call a C/C++ callable library and other functions from within an assembly language program. As discussed in "Calling Assembly Subroutines From C/C++ Programs" on page 1-280, you may want to create a test function to do this in C/C++, and then use the code generated by the compiler as a reference when creating your assembly language program and the argument setup. Using volatile global variables may help clarify the essential code in your test function.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated*. The contents of the scratch registers may be changed without warning by the called C/C++ function. If the assembly language program needs the contents of any of those registers, you *must save* their contents before the call to the C/C++ function and then *restore* those contents after returning from the call.

Use the dedicated registers for their intended purpose only; the compiler, libraries, debugger, and interrupt routines all depend on having a stack available as defined by those registers.

Preserved registers can be used; their contents are not changed by calling a C/C++ function. The function always saves and restores the contents of preserved registers if they are going to change.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. Explore how arguments are passed between an assembly language program and a function by writing a dummy function in C/C++ and compiling it with the `save temporary files` option (the "-save-temps" command-line switch). By examining the contents of volatile global variables in a `*.s` file, you can determine how the C/C++ function passes arguments, and then duplicate that argument setup process in the assembly language program.

The stack must be set up correctly before calling a C/C++ callable function. If you call other functions, maintaining the basic stack model also facilitates the use of the debugger.

The easiest way to do this is to define a C/C++ main program to initialize the run-time system; maintain the stack until it is needed by the C/C++ function being called from the assembly language program; and then continue to maintain that stack until it is needed to call back into C/C++. However, make sure the dedicated registers are correct. You do not need to set the FP prior to the call; the caller's FP is never used by the recipient.

Using Mixed C/C++ and Assembly Naming Conventions

A user should be able to use C/C++ symbols (function names or variable names) in assembly routines and use assembly symbols in C routines. This section describes how to name C/C++ and assembly symbols and shows how to use C/C++ and assembly symbols.

To name an assembly symbol that corresponds to a C/C++ symbol, add an underscore prefix to the C/C++ symbol name when declaring the symbol in assembly. For example, the C/C++ symbol `main` becomes the assembly symbol `_main`.

To use a C/C++ function or variable in your assembly routine, declare it as global in the C/C++ program and import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

The C++ language performs name mangling on function names it defines according to the output and input parameter types of the function. If calling into a C++ defined function from assembly code, the `.EXTERN` symbol needs to be the mangled C++ output name. This is best retrieved by looking at the compiler's assembly output for the C++ source that defines the required function.

To use an assembly function or variable in your C/C++ program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C/C++ program.

Alternatively, the `ccts` compiler provides an "`asm`" linkage specifier (used similarly to the "C" linkage specifier of C++), which when used, removes the need to add an underscore prefix to the symbol that is defined in assembly.

Table 1-40 shows several examples of the C/C++ and assembly interface naming conventions.

Table 1-40. C/C++ Naming Conventions for Symbols

In C/C++ Program	In Assembly Subroutine
`int c_var; /*declared global*/`	`.extern _c_var;`
`void c_func(); /* in C code */`	`.extern _c_func;`
`void cpp_func(void); /* in C++ source */`	`.extern _cpp_func__Fv;`
`extern int asm_var;`	`.global _asm_var;`
`extern void asm_func();`	`.global _asm_func;` `_asm_func:`
`extern "asm" void asm_func();`	`.global asm_func;` `asm_func:`

C++ Programming Examples

This section shows examples of the features specific to C++. These examples are:

- "Using Fract Type Support" on page 1-286
- "Using Complex Number Support" on page 1-287

By default, the `ccts` compiler runs in C mode. To run the compiler in C++ mode, use the appropriate option on the command line, or select the corresponding option in the **Project Options** dialog box in the VisualDSP++ environment.

For example, the following command line

```
ccts -c++ source.cpp -proc ADSP-TS101
```

runs `ccts` with:

`-c++`
Specifies that the following source file is written in ANSI/ISO standard C++ extended with the Analog Devices keywords.

`source.cpp`
Specifies the source file for your program.

`-proc ADSP-TS101`
Specifies that the compiler should produce code suitable for the ADSP-TS101 processor.

Using Fract Type Support

Listing 1-2 on page 1-287 demonstrates the compiler support for the `fract` type and associated arithmetic operators, such as `+` and `*`. The dot product algorithm is expressed using the standard arithmetic operators. The code demonstrates how two variable-length arrays are initialized with fractional literals.

For more information abut the fractional data type and arithmetic, see "C++ Fractional Type Support" on page 1-234.

Listing 1-2. Example Code: Using Fract Data Type — C++ Code

```
#include <fract>
#define N 20
fract x[N] = {.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,
      .5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r,.5r};
fract y[N] = {0,.1r,.2r,.3r,.4r,.5r,.6r,.7r,.8r,.9r,.10r,.1r,
      .2r,.3r,.4r,.5r,.6r,.7r,.8r,.9r};
fract fdot(int n, fract *x, fract *y)
{
   int j;
   fract s;
   s = 0;
   for (j=0; j<n; j++)
   {
      s += x[j] * y[j];
   }
   return s;
}
int main(void)
{
   fdot(N,x,y);
}
```

Using Complex Number Support

The Mandelbrot fractal set is defined by the following iteration on complex numbers:

```
z := z * z + c
```

The `c` values belong to the set for which the above iteration does not diverge to infinity. The canonical set is defined when `z` starts from `zero`.

Listing 1-3 demonstrates the Mandelbrot generator expressed in a simple algorithm using the C++ library `complex` class:

Listing 1-3. Mandelbrot Generator Example — C++ code

```
#include <complex>
#include <stdlib.h>

int iterate (double c, double z, int max)
{
   int n;
   for (n = 0; n<max && abs(z)<2.0; n++)
   {
   z = z * z + c;
   }
   return (n == max ? 0 : n);
}
```

Listing 1-4 shows a C version of the inner computational function of the Mandelbrot generator extracts performance and programming penalties (compared with the C++ version).

Listing 1-4. Mandelbrot Generator Example — C Code

```
int iterate (double creal, double cimag,
double zreal, double zimag, int max)
{
   double real, imag;
   int n;
   real = zreal * zreal;
   imag = zimag * zimag;
   for (n = 0; n<max && (real+imag)<4.5; n++)
   {
   zimag = 2.0 * zreal * zimag + cimag;
   zreal = real - imag + creal;
   real = zreal * zreal;
   imag = zimag * zimag;
   }
   return (n == max ? 0 : n);
}
```

Compiler C++ Template Support

The compiler provides template support C++ templates as defined in the ISO/IEC 14882:1998 C++ standard, with the exception that the export keyword is not supported.

Template Instantiation

Templates are instantiated automatically during compilation using a linker feedback mechanism. This involves compiling files, determining any required template instantiations, and then recompiling those files making the appropriate instantiations. The process repeats until all required instantiations have been made. Multiple recompilations may be required in the case when a template instantiation is made that requires another template instantiation to be made.

By default, the compiler uses a method called *implicit instantiation*, which is common practice, and results in having both the specification and definition available at point of instantiation. This involves placing template specifications in a header ("`.h`") file and the definitions in a source (like "`.cpp`") file. Any file being compiled that includes a header file containing template specifications will instruct the compiler to implicitly include the corresponding "`.cpp`" file containing the definitions of the compiler.

For example, you may have the header file "`tp.h`"

```
template <typename A> void func(A var)
```

and source file "`tp.cpp`"

```
template <typename A> void func(A var)
{
...code...
}
```

Two files "file1.cpp" and "file2.cpp" that include "tp.h" will have file "tp.cpp" included implicitly to make the template definitions available to the compilation.

When generating dependencies, the compiler will only parse each implicitly included .cpp file once. This parsing avoids excessive compilation times in situations where a header file that implicity includes a source file is included several times. If the .cpp file should be included implicitly more than once , the -full-dependency-inclusion switch (on page 1-73) can be used. (For example, the file may contain macro guarded sections of code.) This may result in more time required to generate dependencies.

When generating dependencies, the compiler will only parse each implicitly included .cpp file once. This is to avoid excessive compilation times in situations where a header file that implicity includes a source file is included several times. If the .cpp file should be included implicitly more than once (for example, if the file contains macro guarded sections of code) the -full-dependency-inclusion switch (on page 1-73) can be used. This may result in an increase in time to generate dependencies.

If there is a desire not to use the implicit inclusion method then the switch -no-implicit-inclusion should be passed to the compiler. In the example, we have been discussing, "tp.cpp" will then be treated as a normal source file and should be explicitly linked into the final product.

Regardless of whether implicit instantiation is used or not, the compilation process involves compiling one or more source files and generating a ".ti" file corresponding to the source files being compiled. These ".ti" files are then used by the prelinker to determine the templates to be instantiated. The prelinker creates a ".ii" file and recompiles one or more of the files instantiating the required templates.

The prelinker ensures that only one instantiation of a particular template is generated across all objects. For example, the prelinker ensures that if both "`file1.cpp`" and "`file2.cpp`" invoked the template function with an int, that the resulting instantiation would be generated in just one of the objects.

Identifying Un-instantiated Templates

If for some reason the prelinker is unable to instantiate all the templates that are required for a particular link then a link error will occur. For example:

```
[Error li1021] The following symbols referenced in processor 'P0'
could not be resolved:
    'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
[_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]' refer-
enced from '.\Debug\main.doj'
    'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
[_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
referenced from '.\Debug\main.doj'
    'T1 Complex<T1>::_getReal() const [with T1=short]
[_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
'.\Debug\main.doj'

Linker finished with 1 error
```

Careful examination of the linker errors reveals which instantiations have not been made. Below are some examples.

```
Missing instantiation:
   Complex<short> Complex<short>::conjugate()
Linker Text:
   'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
   [_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]'
   referenced from '.\Debug\main.doj'

Missing instantiation:
   Complex<short> *Buffer<Complex<short>>::getAddress()
```

```
Linker Text:
   'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
   [_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
   referenced from '.\Debug\main.doj'

Missing instantiation:
   Short Complex<short>::getReal()
Linker Text:
   'T1 Complex<T1>::_getReal() const [with T1=short]
   [_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
   '.\Debug\main.doj'
```

There could be many reasons for the prelinker being unable to instantiate these templates, but the most common is that the `.ti` and `.ii` files associated with an object file have been removed. Only source files that can contain instantiated templates will have associated `.ti` and `.ii` files, and without this information, the prelinker may not be able to complete its task. Removing the object file and recompiling will normally fix this problem.

Another possible reason for uninstantiated templates at link time is when implicit inclusion (described above) is disabled but the source code has been written to require it. Explicitly compiling the `.cpp` files that would normally have been implicitly included and adding them to the final link is normally all that is needed to fix this.

Another likely reason for seeing the linker errors above is invoking the linker directly. It is the compiler's responsibility to instantiate C++ templates, and this is done automatically if the final link is performed via the compiler driver. The linker itself contains no support for instantiating templates.

File Attributes

A file attribute is a name-value pair that is associated with a binary object, whether in an object file (`.doj`) or in a library file (`.dlb`). One attribute name can have multiple values associated with it. Attribute names and values are strings. A valid attribute name consists of one or more characters matching the following pattern:

[a-zA-Z_][a-zA-Z_0-9]*

An attribute value is a non-empty character sequence containing any characters apart from NUL.

Attributes help with the placement of code and data. All compiled objects can contain attributes which allow you to place time-critical objects into internal (fast) memory. Using attribute filters in the LDF, you can place objects into internal or external (slow) memory, either individually or in groups.

Automatically-applied Attributes

By default, the compiler automatically applies a number of attributes when compiling a C or C++ file. Figure 1-4 shows a content attribute tree.

The `Content` attributes can be used to map binary objects according to their kind of content, as show by Table 1-41 on page 1-294.

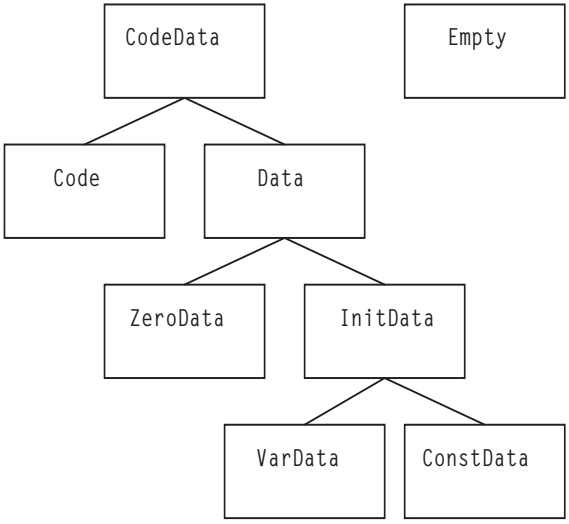

Figure 1-4. Content Attributes

Table 1-41. Values of the `Content` Attribute

Value	Description
`CodeData`	This is the most general value, indicating that the binary object contains a mix of content types.
`Code`	The binary object does not contain any global data, only executable code. This can be used to map binary objects into program memory, or into read-only memory.
`Data`	The binary object does not contain any executable code. The binary object may not be mapped into dedicated program memory. The kinds of data used in the binary object vary.
`ZeroData`	The binary object contains only zero-initialized data. Its contents must be mapped into a memory section with the `ZERO_INIT` qualifier, to ensure correct initialization.
`InitData`	The binary object contains only initialized global data. The contents may not be mapped into a memory section that has the `ZERO_INIT` qualifier.

Table 1-41. Values of the Content Attribute (Cont'd)

Value	Description
VarData	The binary object contains initialized variable data. It must be mapped into read-write memory, and may not be mapped into a memory section with the ZERO_INIT qualifier.
ConstData	The binary object contains only constant data (data declared with the C const qualifier). The data may be mapped into read-only memory (but see also the -const-read-write switch (on page 1-29) and its effects).
Empty	The binary object contains neither functions nor global data.

Default LDF Placement

The default LDF is written so that the order of preference for putting an object in section data or program depends on the value of the prefersMem attribute. Precedence is given in the following order:

1. Highest priority is given to binary objects that have a prefersMem attribute with a value of internal.
2. Next priority is given to binary objects that have no prefersMem attribute, or a prefersMem attribute with a value that is neither internal nor external.
3. Lowest priority is given to binary objects with a prefersMem attribute with the value external.

Sections versus Attributes

File attributes and section qualifiers (on page 1-119) can be thought of as being somewhat similar, since they can both affect how the application is linked. There are important differences, however. These differences will affect whether you choose to use sections or file attributes to control the placement of code and data.

Granularity

Individual components – global variables and functions – in a binary object can be assigned different sections, then those section assignments can be used to map each component of the binary object differently. In contrast, an attribute applies to the whole binary object. This means you do not have as fine control over individual components using attributes as when using sections.

"Hard" versus "soft"

A section qualifier is a *hard* constraint: when the linker maps the object file into memory, it must obey all the section qualifiers in the object file, according to instructions in the LDF. If this cannot be done, or if the LDF does not give sufficient information to map a section from the object file, the linker will report an error.

With attributes, the mapping is *soft*: the default LDFs use the `prefersMem` attribute as a guide to give a better mapping in memory, but if this cannot be done, the linker will not report an error. For example, if there are more objects with `prefersMem=internal` than will fit into internal memory, the remaining objects will spill over into external memory. Likewise, if there are less objects with the attribute `prefersMem!=external` than are needed to fill internal memory, some objects with the attribute `prefersMem=external` may get mapped to internal memory.

Section qualifiers are rules that must be obeyed, while attributes are guidelines, defined by convention, that can be used if convenient and ignored if inconvenient. The `Content` attribute is an example: you can use the `Content` attribute to map `Code` and `ConstData` binary objects into read-only memory, if this is a convenient partitioning of your application. However, you need not do so if you choose to map your application differently.

Number of values

Any given element of an object file is assigned exactly one section qualifier, to determine into which section it should be mapped. In contrast, an object file may have many attributes (or even none), and each attribute may have many different values. Since attributes are optional, and act as guidelines, you need only pay attention to the attributes that are relevant to your application.

Using attributes

You can add attributes to a file in two ways:

- Use `#pragma file_attr` (on page 1-214).
- Use the `-file-attr name` switch (on page 1-34).

Example

Suppose you want the contents of `test.c` to get mapped to external memory by preference. You can do this by adding the following pragma to the top of `test.c`:

```
#pragma file_attr("prefersMem=external")
```

or use the `-file-attr` switch:

```
ccts -file-attr prefersMem=external switches test.c
```

Both of these methods mean that the resulting object file will have the attribute `prefersMem=external`. The LDFs give objects with this attribute the lowest priority when mapping objects into internal memory. As a result, the object is less likely to consume valuable internal memory space, which could be more usefully allocated to another function.

File attributes are used as guidelines rather than rules. If space is available in internal memory after higher-priority objects have been mapped, it is permissible for objects with `prefersMem=external` to be mapped into internal memory.

2 ACHIEVING OPTIMAL PERFORMANCE FROM C/C++ SOURCE CODE

This chapter provides guidance in helping you to tune your application to achieve the best possible code from the compiler. Some implementation choices are available when coding an algorithm, and understanding their impact is crucial to attaining optimal performance.

This chapter contains:

- "General Guidelines" on page 2-3
  provides a four-step basic strategy for designing applications. Also describes topics such as data types, memory usage, and indexed arrays versus pointers
- "Improving Conditional Code" on page 2-28
  provides information about the `expected_true` and `expected_false` built-in functions, which can control the compiler's behavior for specific cases.
- "Loop Guidelines" on page 2-29
  describes in detail how to help the compiler produce the most efficient loop code, including keeping loops short, and avoiding unrolling loops and loop-carried dependencies.
- "Using Built-In Functions in Code Optimization" on page 2-40
  provides information about how to use built-in functions to efficiently use low-level features of the processor hardware while programming in C.

- "Smaller Applications: Optimizing for Code Size" on page 2-44 provides tips and techniques about optimizing the application for full performance and for space.
- "Using Pragmas for Optimization" on page 2-46 describes how to use pragmas to finely tune source code.
- "Useful Optimization Switches" on page 2-54 provides a table listing the compiler switches useful during the optimization process.
- "How Loop Optimization Works" on page 2-55 provides an introduction to some of the concepts used in loop optimization.
- "Assembly Optimizer Annotations" on page 2-72 gives the programmer an understanding of how close to optimal a program is and what more can be done to improve the generated code.

The focus of this chapter is on how to obtain maximal code performance from the compiler. Most of these guidelines also apply when optimizing for minimum code size, although some techniques specific to that goal are also discussed.

The first section looks at some general principles, and how the compiler can lend the most help to your optimization effort. Optimal coding styles are then considered in detail. Special features such as compiler switches, built-in functions, and pragmas are also discussed. The chapter ends with a short example to demonstrate how the optimizer works.

Small examples are included throughout this chapter to demonstrate points being made. Some show recommended coding styles, while others identify styles to be avoided or code that may be possible to improve. These are commented in the code as "GOOD" and "BAD" respectively.

General Guidelines

It is useful to bear in mind the following basic strategy when writing an application:

1. Try to choose an algorithm that is suited to the architecture being targeted. For example, there may be a trade-off between memory usage and algorithm complexity that may be influenced by the target architecture.
2. Code the algorithm in a simple, high-level generic form. Keep the target in mind, especially with respect to choices of data types.
3. You can then turn your attention towards code tuning. For critical code sections, think more carefully about the strengths of the target platform, and make any non-portable changes where necessary.

Choose the language as appropriate.

As a programmer, your first decision is to choose whether to implement your application in C or C++. This decision may be influenced by performance considerations. C++ code using only C features has very similar performance to pure C source. Many higher-level C++ features (for example those resolved at compile-time, such as namespaces, overloaded functions and inheritance) have no performance cost. However, use of some other features may degrade performance, and so the performance loss must be weighed against the richness of expression available in C++. Examples of features that may degrade performance are virtual functions or using classes to implement basic data types.

This section contains:

- "How the Compiler Can Help" on page 2-4
- "Data Types" on page 2-13

- "Getting the Most From IPA" on page 2-16
- "Indexed Arrays Versus Pointers" on page 2-22
- "Function Inlining" on page 2-23
- "Using Inline asm Statements" on page 2-24
- "Memory Usage" on page 2-25

How the Compiler Can Help

The compiler provides many facilities designed to help the programmer including compiler optimizer, PGO, and IPA optimizers.

Using the Compiler Optimizer

There is a vast difference in performance between code compiled optimized and code compiled non-optimized. In some cases, optimized code can run ten or twenty times faster. Optimization should always be used when measuring performance or shipping code as product.

The optimizer in the C/C++ compiler is designed to generate efficient code from source that has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, and hence the greatest freedom to safely manipulate the code. Future releases of the compiler will continue to enhance the optimizer, and expressing algorithms simply will provide the best chance of benefiting from such enhancements.

Note that the default setting (or "debug" mode within the VisualDSP++ IDDE) is for non-optimized compilation in order to assist programmers in diagnosing problems with their initial coding. The optimizer is enabled in VisualDSP++ by checking the **Enable optimization** checkbox under the

Project Options ->Compile tab. This adds the `-O` (enable optimization) switch (on page 1-48) to the compiler invocation. A "release" build from within VisualDSP++ automatically enables optimization.

Using Compiler Diagnostics

There are many features of the C and C++ languages that, while legal, indicate programming errors. There are also aspects that are valid but may be relatively expensive for an embedded environment. The compiler can provide the following diagnostics, which may avoid time and effort characterizing source-related problems:

- Warnings and remarks
- Source and assembly annotations

These diagnostics are particularly important for obtaining high-performance code, since the optimizer aggressively transforms the application to get the best performance, discarding unused or redundant code. If this code is redundant because of a programming error (such as omitting an essential `volatile` qualifier from a declaration), then the code will behave differently from a non-optimized version. Using the compiler's diagnostics may help you identify such situations before they become problems.

Warnings and Remarks

By default, the compiler emits warnings, to the standard error stream at compile-time, when it detects a problem with the source code. Warnings can be disabled individually, with the `-Wsuppress` switch (on page 1-67) or as a class, with the `-w` switch (on page 1-68). However, disabling warnings is inadvisable until each instance has been investigated for problems.

A typical warning involves a variable being used before its value has been set.

Remarks are a lower-severity class of diagnostic. Like warnings, they are produced at compile-time to the standard error stream, but unlike warnings, remarks are suppressed by default. Remarks are typically for situations that are probably correct, but not ideal. Remarks may be enabled as a class with the `-Wremarks` switch (on page 1-67).

A typical remark involves a variable being declared, but never used.

Remarks may be promoted to warnings, through the `-Wwarn` switch (on page 1-67). Both remarks and warnings may be promoted to errors, through the `-Werror` switch (on page 1-67). Here is a procedure for improving overall code quality:

1. Enable remarks with `-Wremarks` and build the application. Gather all warnings and remarks generated.
2. Examine the generated diagnostics and choose those message types that you consider most important. For example, you might select just `cc0223`, a remark that identifies implicitly-declared functions.
3. Promote those remarks and warnings to errors, using the `-Werror` switch (for example, "`-Werror 0223`"), and rebuild the application. The compiler will now fault such cases as errors, so you will have to fix the source to address the issues before your application will build.
4. Once your application builds again, repeat the process, selecting the next group of diagnostics you consider most important.

For example, a typical list of diagnostics might include:

- `cc0223`: function declared implicitly
- `cc0549`: variable used before its value is set
- `cc1665`: variable is possibly used before its value is set, in a loop
- `cc0187`: use of "=" where "==" may have been intended

- `cc1045`: missing return statement at the end of non-void function
- `cc0111`: statement is unreachable

If you have particular cases that are correct for your application, do not let them prevent your application from building because you have raised the diagnostic to an error. For such cases, temporarily lower the severity again within the source file in question by using `#pragma diag` (on page 1-223).

Source and Assembly Annotations

By default, the compiler emits annotations that are embedded in the generated code – either in the object file or in the assembly source, depending on the output form you select. The source-related annotations can be viewed within the IDDE, while the assembly-related annotations give considerably more information about the intricacies of the generated code. Annotations can be used to identify why the compiler has generated code in a particular manner.

For more information, see “Assembly Optimizer Annotations” on page 2-72.

Using the Statistical Profiler

Tuning an application begins with an understanding of which areas of the application are most frequently executed and therefore where improvements would provide the largest gains. The statistical profiling feature provided in VisualDSP++ is an excellent means for finding these areas. More details about how to use it may be found in the *VisualDSP++ 4.5 User's Guide*.

The advantage of statistical profiling is that it is completely unobtrusive. Other forms of profiling insert instrumentation into the code, disturbing the original optimization, code size and register allocation to some degree.

The best methodology is usually to compile with both optimization and debug information generation enabled. You can then obtain a profile of the optimized code while retaining function names and line number information. This gives you accurate results that correspond directly to the C/C++ source. Note that the compiler optimizer may have moved code between lines.

You can obtain an accurate view of your application if you build it optimized but without debug information generation. You can then obtain statistics that relate directly to the assembly code. The only problem with doing this may be in relating assembly lines to the original source. Do not strip out function names when linking, since keeping function names means you can scroll through the assembly window to instructions of interest.

In complex code, you can locate the exact source lines by counting the loops, unless they are unrolled. Looking at the line numbers in the assembly file may also help. (Use the `-save-temps` switch to retain compiler generated assembly files, which have the `.s` filename extension.) The compiler optimizer may have moved code around, so that it does not appear in the same order as in your original source.

Using Profile-Guided Optimization

Profile-Guided Optimization (PGO) is an excellent way to tune the compiler's optimization strategy for the typical run-time behavior of a program. There are many program characteristics that cannot be known statically at compile-time but can be provided by means of PGO. The compiler can use this knowledge to bring about benefits such as more accurate branch-prediction, improved loop transformations, and reduced code-size. The technique is most relevant where the behavior of the application over different data sets is expected to be very similar.

Using Profile-Guided Optimization With a Simulator

The process of using PGO is illustrated in Figure 2-1 on page 2-9.

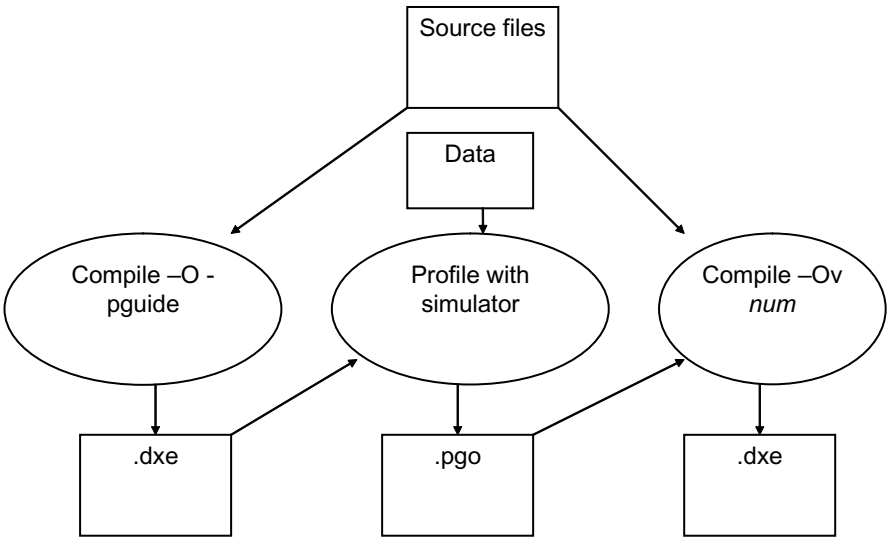

Figure 2-1. PGO Process

1. Compile the application with the `-pguide` switch (on page 1-56) to create an executable file containing the necessary instrumentation for gathering profile data. Enable optimization. For best results, use the `-O` (on page 1-48) or `-ipa` (on page 1-40) switches.
2. Gather the profile. Presently this may only be done using a simulator. To do this, run the executable with one or more training data sets. These training data sets should be representative of the data that you expect the application to process in the field. Note that

unrepresentative training data sets can cause performance degradations when the application is used on real data. The profile is accumulated in a file with the extension `.pgo`.

3. Re-compile the application using this gathered profile data. Place the `.pgo` file on the command line. Optimization should also be enabled at this stage.

Using Profile-Guided Optimization With Non-simulatable Applications

It may not be possible to run a complex application in its entirety under a simulation session (for example, if peripherals not modeled by the simulator are used) meaning PGO is not useable in the conventional way. It may, however, still be possible to benefit from the use of PGO as suggested in the following guidelines.

If the application is structured in a suitably modular fashion, it is possible to extract the core (and, most likely, performance critical) algorithm from the application. A "wrapper" project, which can be run under simulation, can be created that drives input values into the core algorithm, replacing the portions of the application that can not be run under simulation. This project can be used to generate PGO information, which can subsequently be used to optimize the full application. As described previously, it is essential that the input values are representative of real data to achieve best performance.

Leave unmodified as much of the core algorithm as possible, keeping file and function names the same. The `.pgo` files generated from execution of the wrapper project can then be used to optimize the same functions in the full application by including the `.pgo` files in the full application build.

When compiling with a `.pgo` file, the compiler will ignore the data for a function if it detects the function has changed from when the PGO data was generated, and emit a warning.

Profile-Guided Optimization and Multiple Source Uses

In some applications, it is convenient to build the same source file in more than one way, within the same application. For example, a source file might be conditionally-compiled with different macro settings. Alternative, the same file might be compiled once, but linked more than once into the same application, in a multi-core or multi-processor environment. In such circumstances, the typical behaviors of each instance in the application might differ. Therefore, it is desirable to identify the separate instances so that they can be profiled separately and optimized accordingly.

The `-pgo-session` switch (on page 1-55) is used to separate profiles in such cases. It is used during both Stage 1, where the compiler instruments the generated code for profiling, and during Stage 3, where the compiler makes use of gathered profiles to guide the optimization.

During Stage 1, when the compiler instruments the generated code, if the `-pgo-session` switch is used, then the compiler marks the instrumentation as belonging to session `session-id`.

During Stage 3, when the compiler reads gathered profiles, if the `-pgo-session` switch is used, then the compiler ignores all profile data not generated from code that was marked with the same `session-id`.

Therefore, the compiler can optimize each variant of the source's build according to how the variant is used, rather than according to an average of all uses.

Profile-Guided Optimization and the -Ov switch

Note that when a `.pgo` file is placed on the command line, the `-O` optimization switch by default tries to balance between code performance and code-size considerations. It is equivalent to using the `-Ov 50` switch. To optimize solely for performance while using PGO, the switch `-Ov 100`

should be used. The `-Ov n` switch (on page 1-50) is discussed further when looking at the specific issues involved in optimization for space. (See "Smaller Applications: Optimizing for Code Size" on page 2-44.)

When to use Profile-Guided Optimization

PGO should always be performed as the last optimization step. If the application source code is changed after gathering profile data, this profile data becomes invalid. The compiler does not use profile data when it can detect that it is inaccurate. However, it is possible to change source code in a way that is not detectable to the compiler (for example, by changing constants). The programmer should ensure that the profile data used for optimization remains accurate.

For more details on PGO, refer to "Optimization Control" on page 1-81.

Using Interprocedural Optimization

To obtain the best performance, the optimizer often requires information that can only be determined by looking outside the function that it is working on. For example, it helps to know what data can be referenced by pointer parameters, or whether a variable actually has a constant value.

The `-ipa` compiler switch (on page 1-40) enables interprocedural analysis (IPA), which can make this available. When this switch is used, the compiler is called again from the link phase to recompile the program using additional information obtained during previous compilations.

Because it only operates at link time, the effects of IPA are not seen if you compile with the `-S` switch (on page 1-59). To see the assembly file when IPA is enabled, use the `-save-temps` switch (on page 1-59), and look at the `.s` file produced after your program has been built.

As an alternative to IPA, you can achieve many of the same benefits by adding pragma directives and other declarations such as `__builtin_aligned` to provide information to the compiler about how each function interacts with the rest of the program.

These directives are further described "Using __builtin_aligned" on page 2-18 and "Using Pragmas for Optimization" on page 2-46.

Data Types

Table 2-1 shows the following scalar data types that the compiler supports.

Table 2-1. Scalar Data Types

Single-Word Fixed-Point Data Types: Native Arithmetic	
`char`	32-bit signed integer (default) 8-bit signed integer (with `-char-size-8`)
`unsigned char`	32-bit unsigned integer (default) 8-bit unsigned integer (with `-char-size-8`)
`short`	32-bit signed integer (default) 16-bit signed integer (with `-char-size-8`)
`unsigned short`	32-bit unsigned integer (default) 16-bit unsigned integer (with `-char-size-8`)
`int`	32-bit signed integer
`unsigned int`	32-bit unsigned integer
`long`	32-bit signed integer
`unsigned long`	32-bit unsigned integer
`long long`	64-bit signed integer
`unsigned long long`	64-bit unsigned integer
Floating-Point Data Types: Native Arithmetic	
`float`	32-bit floating-point

Table 2-1. Scalar Data Types

`double`	32-bit floating-point (default
Floating-Point Data Types: Emulated Arithmetic	
`double`	64-bit floating-point (set with the `-double-size-64` switch)
`long double`	64-bit floating-point

Fractional data types are represented using the integer types. Manipulation of these is best done by use of built-in functions, described in "System Support Built-in Functions" on page 2-41.

Avoiding Emulated Arithmetic

Arithmetic operations for some types are implemented by library functions because the processor hardware does not directly support these types. Consequently, operations for these types are far slower than native operations—sometimes by a factor of a hundred—and also produce larger code. These types are marked as "Emulated Arithmetic" in "Data Types" on page 2-13.

The hardware does not provide direct support for division, so division and modulus operations are almost always multi-cycle operations, even on integral type inputs. An exception is that the compiler converts an integral division by a power of two to a right-shift operation if the value of the divisor is known, albeit with a few fix-up instructions if the types are signed. In this case it is worth considering if an unsigned division could be used. If the compiler has to issue a full division operation, it usually needs to generate a call to a library function.

In addition, although most arithmetic operations on `long long` integers are supported by the hardware, multiplication is not. However the `long long` type is often useful for bit manipulation algorithms because of the number of bits that can be processed in each operation.

When the compiler has to generate a call to a library function for one of these arithmetic operators that are not supported by the hardware, performance suffers not only because the operation takes multiple cycles, but also because the effectiveness of the compiler optimizer is reduced.

For example, such an operation in a loop can prevent the compiler from making use of efficient zero-overhead hardware loop instructions. Also, calling the library to perform the required operation can change values held in scratch registers before the call, so the compiler has to generate more stores and loads from the data stack to keep values required after the call returns. Emulated arithmetic operators should therefore be avoided where possible, especially in loops.

Using Sub-Word Types with Caution

The size of `char` and `short` integer types may be changed through the `-char-size-8` switch (on page 1-28). Although the TigerSHARC hardware does not support 8- or 16-bit loads and stores from memory, they can be simulated in code. However, it is important to remember that they are considerably less efficient than full-word memory accesses.

Sub-word-sized data may permit greater throughput on highly "paralellizable" inner loops, but sub-word-sized scalars or use in "non-paralellizable" loops, degrade the performance. Therefore, use sub-word-sized integers with caution, or where compatibility with true byte-addressable architectures is required.

However, the TigerSHARC architecture does provide hardware support for data types such as short vectors of 16-bit and 8-bit `ints` and 16-bit fixed-point `complex`. Though not directly supported by the compiler, access to the instructions is available through intrinsic (built-in) functions.

Note that these 16-bit operations are often more efficient than 32-bit ones. However, the compiler does not generate a 16-bit operation where you wrote a 32-bit operations even if it is obvious that the 16-bit operation would generate the same result. Apart from the `-char-size-8` switch,

the only way to get 16-bit operations is to use an built-in function. For more information on supported built-in functions, refer to "Compiler Built-In Functions" on page 1-126. Even if you use the `-char-size-8` switch, intrinsics can be a more reliable way to ensure maximum throughput in critical loops.

The following code example shows scalar product written with 4x16 short vector intrinsics. It illustrates the approved style for code written with intrinsic functions.

```
// GOOD: uses 4x16 short vector intrinsics
   typedef long long int4x16;

   #define add(x,y) __builtin_add_4x16(x,y)
   #define mult(x,y) __builtin_mult_i4x16(x,y)
   #define sum(x) __builtin_sum_4x16(x)
   int sp4x16(int4x16 a[], int4x16 b[], int n)
   {

      int i;
      int4x16 sum4 = 0;
      for (i = 0; i < n/4; ++i)
          sum4 = add(sum4, mult(a[i], b[i]));
      return sum(sum4);
   }
```

Getting the Most From IPA

Interprocedural analysis (IPA) is designed to try to propagate information about the program to parts of the optimizer that can use it. This section looks at what information is useful, and how to structure your code to make this information easily accessible to the analysis.

Initialize Constants Statically

IPA identifies variables that have only one value and replaces them with constants, resulting in a host of benefits for the optimizer's analysis. For this to happen, a variable must have a single value throughout the program. If the variable is statically initialized to zero, as all global variables are by default, and is subsequently assigned some other value at another point in the program, then the analysis sees two values and does not consider this variable to have a constant value. For example,

```
// BAD: IPA cannot see that val is a constant
   #include <stdio.h>
   int val;           // initialized to zero

   void init() {
      val = 3;        // re-assigned
   }

   void func() {
      printf("val %d",val);
   }

   int main() {
      init();
      func();
   }
```

The code is better written as

```
// GOOD: IPA knows val is 3
   #include <stdio.h>
   const int val = 3;    // initialized once

   void init() {
   }

   void func() {
      printf("val %d",val);
   }

   int main() {
      init();
```

```
    func();
}
```

Quad-Word-Aligning Your Data

To make most efficient use of the hardware, it must be kept fed with data. In many algorithms, the balance of data accesses to computations is such that, to keep the hardware fully utilized, data must be fetched with loads wider than 32 bits.

The hardware requires that references to memory be naturally aligned. Therefore, 64-bit references must be at even address locations, and 128-bit references at quad-word-aligned addresses. For the most efficient code to be generated, ensure that data buffers are quad-word-aligned.

The compiler helps to establish the alignment of array data. The stack frames are kept quad-word-aligned. Top-level arrays are allocated at quad-word-aligned addresses, regardless of their data types. However, arrays within structures are not aligned beyond the required alignment for their type. It may be worth using the `#pragma align n` directive to force the alignment of arrays in this case.

If you write programs that pass only the address of the first element of an array as a parameter, and loops that process these input arrays an element at a time, starting at element zero, then IPA should be able to establish that the alignment is suitable for full-width accesses.

Where an inner loop processes a single row of a multi-dimensional array, try to ensure that each row begins on a word boundary. This may require the insertion of dummy data at the end of each row to make the row length a multiple of four words.

Using __builtin_aligned

To avoid the need to use IPA to propagate alignment, and for situations when IPA cannot guarantee the alignment (but you can), use the `__builtin_aligned` function to assert the alignment of important point-

ers, meaning that the pointer points to data that is aligned. Remember when adding this declaration that you are responsible for making sure it is valid, and that if the assertion is not true, the code produced by the compiler is likely to malfunction.

The assertion is particularly useful for function parameters, although you may assert that any pointer is aligned. For example, when compiling the function:

```
// BAD: without IPA, compiler does not know the alignment of a
and b
   void copy(char *a, char *b) {
      int i;
      for (i=0; i<100; i++)
          a[i] = b[i];
   }
```

the compiler does not know the alignment of pointers `a` and `b` if IPA is not being used. However, by modifying the function to the following:

```
// GOOD: Both pointer parameters are known to be aligned
   void copy(char *a, char *b) {
      int i;
      __builtin_aligned(a, 4);
      __builtin_aligned(b, 4);
      for (i=0; i<100; i++)
          a[i] = b[i];
   }
```

the compiler is told that the pointers are aligned on quad-word boundaries. To assert instead that both pointers are always aligned one `char` before a quad-word boundary, use:

```
 // GOOD: Both pointer parameters are known to be misaligned
   void copy(char *a, char *b) {
   int i;
   __builtin_aligned(a+1, 4);
   __builtin_aligned(b+1, 4);
   for (i=0; i<100; i++)
```

```
        a[i] = b[i];
    }
```

The expression used as the first parameter to the built-in function obeys the usual C rules for pointer arithmetic. However, the second parameter to the built-in function should give the alignment in words. The exception is that when using the `-char-size-8` option, the alignment should instead be given in bytes. In that case, to specify that a pointer `a` is quad-word aligned, use `__builtin_aligned(a, 16)`.

When a loop has up to two non-aligned pointers, the compiler can make use of the hardware data alignment buffers to access 128-bits of data from those pointers. Stores to memory via misaligned pointers are more difficult, so it is more important to store to an aligned buffer than to load from one.

Avoiding Aliases

It may seem that the iterations may be performed in any order in the following loop:

```
// BAD: a and b may alias each other
   void fn(char a[], char b[], int n) {
      for (i = 0; i < n; ++i)
          a[i] = b[i];
   }
```

but `a` and `b` are both parameters, and, although they are declared with `[]`, they are in fact pointers, which may point to the same array. When the same data may be reachable through two pointers, they are said to alias each other.

If the `-ipa` switch is enabled, the compiler looks at the call sites of `fn` and try to determine whether `a` and `b` can ever point to the same array.

Even with IPA, it is easy to create what appears to the compiler as an alias. The analysis works by associating pointers with sets of variables that they may refer to some point in the program. If the sets for two pointers intersect, then both pointers are assumed to point to the union of the two sets.

If `fn` above were called in two places with global arrays as arguments, then IPA would have the results shown below:

```
   // GOOD: sets for a and b do not intersect: a and b are not
aliases
   fn(glob1, glob2, N);
   fn(glob1, glob2, N);

   // GOOD: sets for a and b do not intersect: a and b are not
aliases
   fn(glob1, glob2, N);
   fn(glob3, glob4, N);

 // BAD: sets intersect; a and b may be aliases
   fn(glob1, glob2, N);
   fn(glob3, glob1, N);
```

The third case arises because IPA considers the union of all calls at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account and the number of permutations would make compilation time impracticably long.

The lack of control flow analysis can also create problems when a single pointer is used in multiple contexts. For example, it is better to write:

```
 // GOOD: p and q do not alias
   int *p = a;
   int *q = b;
      // some use of p
      // some use of q
```

.

than

```
// BAD: uses of p in different contexts may alias
  int *p = a;
     // some use of p
  p = b;
     // some use of p
```

because the latter may cause extra apparent aliases between the two uses.

Indexed Arrays Versus Pointers

The C language allows a program to access data from an array in two ways: either by indexing from an invariant base pointer, or by incrementing a pointer. The following two versions of vector addition illustrate the two styles.

Style 1: using indexed arrays

```
void va_ind(const short a[], const short b[], short out[], int n) {
   int i;
   for (i = 0; i < n; ++i)
       out[i] = a[i] + b[i];
}
```

Style 2: using pointers

```
void va_ptr(const short a[], const short b[], short out[], int n) {
   int i;
   short *pout = out;
   const short *pa = a, *pb = b;
   for (i = 0; i < n; ++i)
       *pout++ = *pa++ + *pb++;
}
```

Trying Pointer and Indexed Styles

One might hope that the chosen style would not make any difference to the generated code, but this is not always the case. Sometimes, one version of an algorithm generates better optimized code than the other, but it is not always the same style that is better.

Try both pointer and index styles.

The pointer style introduces additional variables that compete with the surrounding code for resources during the compiler optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler—sometimes it does not do the job as well as you could do by hand.

The best strategy is to start with array notation. If the generated code looks unsatisfactory, try using pointers. Outside the critical loops, use the indexed style, since it is easier to understand.

Function Inlining

The function inlining may be used in two ways:

- By annotating functions in the source code with the `inline` keyword. In this case, function inlining is performed only when optimization is enabled.
- By turning on automatic inlining with the `-Oa` switch (on page 1-49). This switch automatically enables optimization.

Inline small, frequently executed functions.

You can use the compiler's `inline` keyword to indicate that functions should have code generated inline at the point of call. Doing this avoids various costs such as program flow latencies, function entry and exit instructions and parameter passing overheads.

Using an `inline` function also has the advantage that the compiler can optimize through the inline code and does not have to assume that scratch registers and condition states are modified by the call. Prime candidates for inlining are small, frequently-used functions because they cause the least code-size increase while giving most performance benefit.

As an example of the usage of the `inline` keyword, the function below sums two input parameters and returns the result.

```
// GOOD: use of the inline keyword
  inline int add(int a, int b) {
     return (a+b);
  }
```

Inlining has a code size-to-performance trade-off that should be considered when it is used. With `-Oa`, the compiler automatically inlines small functions where possible. If the application has a tight upper code-size limit, the resulting code-size expansion may be too great. It is worth considering using automatic inlining in conjunction with the `-Ov num` switch (on page 1-50) to restrict inlining (and other optimizations with a code-size cost) to parts of the application that are performance-critical. This is considered in more detail later in this chapter.

Using Inline asm Statements

The compiler allows use of inline `asm` statements to insert small sections of assembly code into C code.

Avoid use of inline `asm` statements where built-in functions may be used instead

The compiler does not intensively optimize code that contains inline `asm` statements because it has little understanding about what the code in the statement does. In particular, use of an `asm` statement in a loop may inhibit useful transformations.

The compiler has been enhanced with a large number of built-in functions. These functions generate specific hardware instructions and are designed to allow the programmer to more finely tune the code produced by the compiler, or to allow access to system support functions. A complete list of compiler's built-in functions is given in "Compiler Built-In Functions" on page 1-126.

Use of these built-in functions is much preferred to using inline `asm` statements. Since the compiler knows what each built-in function does, it can easily optimize around them. Conversely, since the compiler does not parse `asm` statements, it does not know what they do, and therefore is hindered in optimizing code that uses them. Note also that errors in the text string of an `asm` statement are caught by the assembler and not the compiler.

Memory Usage

The compiler, in conjunction with the use of the linker description file (`.LDF`), allows the programmer control over where data is placed in memory. This section describes how to best lay out data for maximum performance.

Putting Arrays into Different Memory Sections

The DSP hardware can support two memory operations on a single instruction line, combined with a compute instruction. However, two memory operations complete in one cycle only if the two addresses are situated in different memory blocks; if both access the same block, then a stall will be incurred.

Try to put arrays into different memory sections.

Take as an example the dot product loop below. Because data is loaded from both array `a` and array `b` in every iteration of the loop, it may be useful to ensure that these arrays are located in different blocks.

```
 // BAD: compiler assumes that two memory accesses together may
give a stall
   for (i=0; i<100; i++)
      sum += a[i] * b[i];
```

The efficient memory usage is facilitated using the "Dual Memory Support Language Keywords" compiler extension (see "Memory Support Keywords (pm dm)" on page 1-115). Placing a `pm` qualifier before the type definition tells the compiler that the array is located in what is notionally called "Program Memory" (`pm`).

The memory of a TigerSHARC processor is one unified address space and there is no restriction concerning in which part of memory program code or data can be placed into. However, the default `.LDF` files ensure that `pm`-qualified data is placed in a different memory block than non-qualified (or `dm`-qualified) data, thus allowing two accesses to occur simultaneously without incurring a stall.

The array declaration of one of either `a` or `b` is modified to

```
pm int a[100];
```

and any pointers to the buffer `a` become, for example,

```
pm int *p = a;
```

It is also possible to use the `.LDF` file to place data in different user-defined memory sections. This has the advantage that it is possible to define more than two sections which do not conflict with each other. Consider again the example above.

First, let's define two memory banks in the `MEMORY` portion of the `.LDF` file, such as in the following example.

Example: `MEMORY` portion of the `.LDF` file modified to define memory banks.

```
MEMORY {
   BANK_A1 {
      TYPE(RAM) WIDTH(32)
      START(start_address_1) END(end_address_1)
   }
   BANK_A2 {
      TYPE(RAM) WIDTH(32)
      START(start_address_2) END(end_address_2)
   }
}
```

Then, configure the `SECTIONS` portion to instruct the linker to place data sections in specific memory banks.

Example: `SECTIONS` portion of the `.LDF` file modified to define memory banks.

```
SECTIONS {
   bank_a1 {
      INPUT_SECTION_ALIGN(4)
      INPUT_SECTIONS($OBJECTS(bank_a1))
   } >BANK_A1
   bank_a2 {
      INPUT_SECTION_ALIGN(4)
      INPUT_SECTIONS($OBJECTS(bank_a2))  } >BANK_A2
}
```

In the C source code, declare arrays with the `section("section_name")` construct preceding a buffer declaration; in this case,

```
section("bank_a1") short a[100];
section("bank_a2") short b[100];
```

This ensures that the two array accesses may occur simultaneously without incurring a stall.

Any pointers to the buffers `a` and `b` should likewise be modified:

```
section("bank_a1") short *p = a;
```

Note that the explicit placement of data in sections or in Program Memory can only be done for global data.

Using the Bank Qualifier

The bank qualifier can also be useful to write functions which make use of the fact that buffers are placed in separate memory blocks. For example, it might be useful to create a function

```
// GOOD: uses bank qualifier to allow simultaneous access to p
and q
   void func(int bank("red") *p, int bank("blue") *q) {
        // some code
   }
```

if you would like to call `func` in different places, but always with pointers to buffers in different sections of memory. The bank qualifier symbolizes that the buffers are in different sections without requiring that the sections themselves be specified.

Therefore, `func` (in the example above) may be called with the first parameter pointing to memory in `section("bank_a1")` and the second pointing to data in `section("bank_a2")` or vice versa.

Improving Conditional Code

When compiling conditional statements, the compiler attempts to predict whether the condition will usually evaluate to true or to false, and will arrange for the most efficient path of execution to be that which is expected to be most commonly executed.

You can use the `expected_true` and `expected_false` built-in functions to control the compiler's behavior for specific cases. By using these functions, you can tell the compiler which way a condition is most likely to evaluate, and so influence the default flow of execution. For example,

```
if (buffer_valid(data_buffer))
   if (send_msg(data_buffer))
      system_failure();
```

shows two nested conditional statements. If it was known that, for this example, `buffer_valid()` would usually return true, but that `send_msg()` would rarely do so, the code could be written as

```
if (expected_true(buffer_valid(data_buffer)))
   if (expected_false(send_msg(data_buffer)))
      system_failure();
```

See "Optimization Guidance Built-in Functions" on page 1-129 (on `expected_true` and `expected_false` functions) for more information.

The compiler can also determine the most commonly-executed branches automatically, using Profile-Guided Optimization. See "Optimization Control" on page 1-81 for more details.

Loop Guidelines

Loops are where an application ordinarily spends the majority of its time. It is therefore useful to look in detail at how to help the compiler to produce the most efficient code possible for them.

Keeping Loops Short

For best code efficiency, loops should be as small as possible. Large loop bodies are usually more complex and difficult to optimize. Additionally, they may require register data to be stored in memory. This causes a decrease in code density and execution performance.

Avoiding Unrolling Loops

Do not unroll loops yourself.

Not only does loop unrolling make the program harder to read but it also prevents optimization by complicating the code for the compiler.

```
// GOOD: the compiler unrolls if it helps
   void va1(const short a[], const short b[], short c[], int n)
   {
      int i;
      for (i = 0; i < n; ++i) {
          c[i] = b[i] + a[i];
      }
   }

// BAD: harder for the compiler to optimize
   void va2(const short a[], const short b[], short c[], int n)
   {
      short xa, xb, xc, ya, yb, yc;
      int i;
      for (i = 0; i < n; i+=2) {
          xb = b[i]; yb = b[i+1];
          xa = a[i]; ya = a[i+1];
          xc = xa + xb; yc = ya + yb;
          c[i] = xc; c[i+1] = yc;
      }
   }
```

Avoiding Loop-Carried Dependencies

A loop-carried dependency exists when computations in a given iteration of a loop cannot be completed without knowledge of the values calculated in earlier iterations. When a loop has such dependencies, the compiler cannot overlap loop iterations. Some dependencies are caused by scalar variables that are used before they are defined in a single iteration. However, an optimizer can reorder iterations in the presence of the class of

scalar dependencies known as *reductions*. These are loops that reduce a vector of values to a scalar value using an associative and commutative operator—a common case being multiply and accumulate.

```
// BAD: loop-carried dependence is a scalar dependency
   for (i = 0; i < n; ++i)
       x = a[i] - x;

// GOOD: loop-carried dependence is a reduction
   for (i = 0; i < n; ++i)
       x += a[i] * b[i];
```

In the first case, the scalar dependency is the subtraction operation; the variable `x` is modified in a manner that gives different results if the iterations are performed out of order. In contrast, in the second case the properties of the addition operator stipulate that the compiler can perform the iterations in any order and still get the same result. Other examples of operators that are reductions are `bitwise` and/or, and `min/max`. In particular, the existence of loop-carried dependencies that are not reductions is one criterion that prevents the compiler from being able to vectorize a loop—that is, to execute more than one iteration concurrently.

Floating-point addition is by default assumed to obey the criteria necessary to be considered a reduction. However, strictly speaking, rounding effects can change the result when the order of summation is varied. Behavior consistent with the original program can be regained using the `-no-fp-associative` compiler switch (see on page 1-46).

Avoiding Loop Rotation by Hand

Do not rotate loops by hand.

Programmers are often tempted to “rotate” loops in processor code by “hand” attempting to execute loads and stores from earlier or future iterations at the same time as computation from the current iteration. This

technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. However, it is better to give the compiler a “normalized” version, and leave the rotation to the compiler.

For example,

```
// GOOD: is rotated by the compiler
   int ss(short *a, short *b, int n) {
      int sum = 0;
      int i;
      for (i = 0; i < n; i++) {
          sum += a[i] + b[i];
      }
      return sum;
   }

// BAD: rotated by hand—hard for the compiler to optimize
   int ss(short *a, short *b, int n) {
      short ta, tb;
      int sum = 0;
      int i = 0;
      ta = a[i]; tb = b[i];
      for (i = 1; i < n; i++) {
          sum += ta + tb;
          ta = a[i]; tb = b[i];
      }
      sum += ta + tb;
      return sum;
   }
```

By rotating the loop, the scalar variables `ta` and `tb` have been added, introducing loop-carried dependencies.

Avoiding Array Writes in Loops

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from a reads a value defined on a previous iteration or one that is overwritten in a subsequent iteration.

```
// BAD: has array dependency
   for (i = 0; i < n; ++i)
       a[i] = b[i] * a[c[i]];
```

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as “induction variables”.

```
// GOOD: uses induction variables
   for (i = 0; i < n; ++i)
       a[i+4] = b[i] * a[i];
```

Inner Loops Versus Outer Loops

Inner loops should iterate more than outer loops.

The optimizer focuses on improving the performance of inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop if it is going to make the loop body run faster.

Therefore, try to make sure that your algorithm also spends most of its time in the inner loop; otherwise it may actually be made to run slower by optimization. If you have nested loops (where the outer loop runs many times and the inner loop runs a small number of times), it may be possible to rewrite the loops so that the outer loop has fewer iterations.

Avoiding Conditional Code in Loops

If a loop contains conditional code, control-flow latencies may incur large penalties if the compiler has to generate conditional jumps within the loop. In some cases, the compiler is able to convert `IF-THEN-ELSE` and `?` constructs into conditional instructions. In other cases, it is able to relocate the expression evaluation outside of the loop entirely. However, for important loops, linear code should be written where possible.

There are several techniques that can be used to try to remove the necessity for conditional code. For example, there is hardware support for `min` and `max`. The compiler usually succeeds in transforming conditional code equivalent to `min` or `max` into the single instruction. With particularly convoluted code the transformation may be missed, in which case it is better to use `min` or `max` in the source code.

The compiler does not perform the loop transformation that interchanges conditional code and loop structures. Instead of writing

```
// BAD: loop contains conditional code
  for (i=0; i<100; i++) {
     if (mult_by_b)
        sum1 += a[i] * b[i];
     else
        sum1 += a[i] * c[i];
  }
```

it is better to write

```
// GOOD: two simple loops can be optimized well
  if (mult_by_b) {
     for (i=0; i<100; i++)
         sum1 += a[i] * b[i];
  } else {
     for (i=0; i<100; i++)
         sum1 += a[i] * c[i];
  }
```

if this is an important loop.

Avoiding Placing Function Calls in Loops

The compiler usually is unable to generate a hardware loop if the loop contains a function call due to the expense of saving and restoring the context of a hardware loop. In addition to obvious function calls, such as `printf()`, hardware loop generation can also be prevented by operations such as division, modulus, and some type coercions.

These operations may require implicit calls to library functions. For more details, see "Data Types" on page 2-13.

Avoiding Non-Unit Strides

If you write a loop, such as

```
// BAD: non-unit stride means division may be required
   for (i=0; i<n; i+=3) {
         // some code
   }
```

then for the compiler to turn this into a hardware loop, it needs to work out the loop trip count. To do so, it must divide `n` by 3. The compiler decides that this is worthwhile as it speeds up the loop, but division is an expensive operation. Try to avoid creating loop control variables with strides of non-unit magnitude.

In addition, try to keep memory accesses in consecutive iterations of an inner loop contiguous. This is particularly applicable to multi-dimensional arrays. Therefore,

```
// GOOD: memory accesses contiguous in inner loop
   for (i=0; i<100; i++)
      for (j=0; j<100; j++)
          sum += a[i][j];
```

is likely to be better than

```
// BAD: loop cannot be unrolled to use wide loads
   for (i=0; i<100; i++)
      for (j=0; j<100; j++)
          sum += a[j][i];
```

as the former is more amenable to vectorization.

Similarly, two-dimensional arrays should be defined in a single block of memory rather than as an array of pointers to rows, all separately allocated with `malloc`. It is difficult for the compiler to keep track of the alignment of the pointers of the latter.

Loop Control

Use `int` types for loop control variables and array indices. Use automatic variables for loop control and loop exit test.

For loop control variables and array indices, it is always better to use signed `int`s rather than any other integral type. The C standard requires various type promotions and standard conversions that complicate the code for the compiler optimizer. Frequently, the compiler is still able to deal with such code and create hardware loops and pointer induction variables. However, it does make it more difficult for the compiler to optimize and may occasionally result in under-optimized code.

The same advice goes for using automatic (local) variables for loop control. It is easy for a compiler to see that an automatic scalar, whose address is not taken, may be held in a register during a loop. But it is not as easy when the variable is a global or a function static.

Therefore, code such as

```
// BAD: may need to reload globvar on every iteration
   for (i=0; i<globvar; i++)
       a[i] = 10;
```

may not create a hardware loop if the compiler cannot be sure that the write into the array `a` does not change the value of the global variable. The `globvar` variable must be re-loaded each time around the loop before performing the exit test.

In this circumstance, the programmer can make the compiler's job easier by writing:

```
// GOOD: easily becomes hardware loop
   int upper_bound = globvar;
   for (i=0; i<upper_bound; i++)
       a[i] = 10;
```

Using the Restrict Qualifier

The `restrict` qualifier provides one way to help the compiler resolve pointer aliasing ambiguities. Accesses from distinct `restricted` pointers do not interfere with each other. The loads and stores in the following loop

```
// BAD: possible alias of arrays a and b
   for (i=0; i<100; i++)
       a[i] = b[i];
```

may be disambiguated by writing

```
// GOOD: restrict qualifier tells compiler that memory accesses
do not alias
   int * restrict p = a;
   int * restrict q = b;
   for (i=0; i<100; i++)
       *p++ = *q++;
```

The `restrict` keyword is particularly useful on function parameters. The `restrict` keyword currently has no effect when used on variables defined in a scope nested within the outermost scope of a function.

Using the Const Qualifier

By default, the compiler assumes that the data referenced by a pointer to `const` type does not change. Therefore, another way to tell the compiler that the two arrays a and b do not overlap is to use the `const` keyword.

```
// GOOD: pointers disambiguated via const qualifier
   void copy(short *a, const short *b) {
      int i;
      for (i=0; i<100; i++)
          a[i] = b[i];
   }
```

The use of `const` in the above example has a similar effect on the `no_alias` pragma (see in "#pragma no_alias" on page 2-53). In fact, the `const` implementation is better since it also allows the optimizer to use the fact that accesses via a and b are independent in other parts of the code, not just the inner loop.

In C, it is legal, though bad programming practice, to use casts to allow the data pointed to by pointers to `const` type to change. This should be avoided since, by default, the compiler generates code that assumes `const` data does not change. If you have a program that modifies `const` data through a pointer, you can generate standard-conforming code by using the compile-time flag `-const-read-write` (see on page 1-29).

Avoiding Long Latencies

All pipelined machines introduce stall cycles when you cannot execute the current instruction until a prior instruction has exited the pipeline. For example, the TigerSHARC processor stalls for four cycles on a table lookup. `a[b[i]]` takes four cycles more than expected.

If a stall is seen empirically, but it is not obvious to you exactly why it is occurring, a good way to learn about the cause is the **Pipeline Viewer**. This can be accessed through **Debug Windows -> Pipeline Viewer** in the

VisualDSP++ 4.0 IDDE. By single-stepping through the program, you may see where the stall occurs. Note that the **Pipeline Viewer** is only available within a simulator session.

Using Built-In Functions in Code Optimization

Built-in functions, also known as compiler intrinsics, provide a method for the programmer to efficiently use low-level features of the DSP hardware while programming in C. Although this section does not cover all the built-in functions available (for more information, refer to "Compiler Built-In Functions" on page 1-126), it presents some code examples where implementation choices are available to the programmer.

Using Fractional Data

Fractional data, represented as an integral type, can be manipulated in two ways: the first way involves the use of long promoted shifts and multiply constructs. The second way involves the use of compiler built-in functions. The built-in functions are recommended, as they give you the most control over your data. Consider the fractional dot product algorithm. This may be written as:

```
// BAD: uses shifts to implement fraction multiplication
   short dot_product (short *a, short *b) {
      int i;
      short sum=0;
      for (i=0; i<100; i++) {
          /* this line is performance critical */
          sum += ((a[i]*b[i]) >> 15);
      }
      return sum;
   }
```

This presents some problems to the optimizer. Normally, the code generated here would be a multiply, followed by a shift, then an accumulation. However, the processor hardware has a fractional multiply/accumulate instruction that performs all these tasks in one cycle.

The recommended coding style is to use compiler intrinsics (built-in functions). In the following example, an intrinsic is used to multiply fractional 16-bit data. Note that it is also assumed that it has been possible to rearrange the 16-bit fractional data to be packed two items per word.

```
// GOOD: uses intrinsics to implement fractional multiplication
   #include <math.h>
   short dot_product(int4x16 *a, int4x16 *b) {
      int i;
      int4x16 sum=0;
      for (i=0; i<100; i++) {
          /* this line is performance critical */
          sum = __builtin_add_4x16(
                  sum, __builtin_mult_fr4x16(a[i],b[i]));
      }
      return __builtin_sum_4x16(sum);
   }
```

Note that the `int4x16` type, along with other related ones, are merely `typedefs` to C integer types used by convention in standard include files. The compiler does not have any in-built knowledge of these types and treats them exactly as the integer types that they are `typedefed` to.

System Support Built-in Functions

Built-in functions allow to perform low-level system management, in particular for the manipulation of system registers (defined in `sysreg.h`). It is usually better to use these built-in functions rather than inline `asm` statements.

The built-in functions cause the compiler to generate efficient inline instructions and their use often results in better optimization of the surrounding code at the point where they are used. Using built-in functions also usually results in improved code readability. For more information on supported built-in functions, refer to "Compiler Built-In Functions" on page 1-126.

Using Circular Buffers

Circular buffers are often extremely useful in DSP-style code. They can be used in several ways. Consider the C code:

```
// GOOD: the compiler knows that b is accessed as a circular
buffer
   for (i=0; i<1000; i++) {
       sum += a[i] * b[i%20];
   }
```

Clearly the access to array `b` is a circular buffer. When optimization is enabled, the compiler produces a hardware circular buffer instruction for this access.

However, consider the slightly more complicated example:

```
// BAD: may not be able to use circular buffer to access b
   for (i=0; i<1000; i+=n) {
       sum += a[i] * b[i%20];
   }
```

In this case, the compiler does not know if `n` is positive and less than 20. If it is, then the access may be correctly implemented as a hardware circular buffer. On the other hand, if it is greater than 20, a circular buffer increment may not yield the same results as the C code.

The programmer has two options here. One is to compile with the `-force-circbuf` switch (on page 1-35). This tells the compiler that any access of the form `a[i%n]` should be considered as a circular buffer. Before using this switch, check that this assumption is valid for your application.

The second, and preferred option, is to use built-in functions to perform the circular buffering. Two functions (`__builtin_circindex` and `__builtin_circptr`) are provided for this purpose.

To make it clear to the compiler that a circular buffer should be used, you may write either:

```
// GOOD: explicit use of circular buffer via __builtin_circindex
   for (i=0, j=0; i<1000; i+=n) {
       sum += a[i] * b[j];
       j = __builtin_circindex(j, n, 20);
   }
```

or

```
// GOOD: explicit use of circular buffer via __builtin_circptr
   int *p = b;
   for (i=0, j=0; i<1000; i+=n) {
       sum += a[i] * (*p);
       p = __builtin_circptr(p, n, b, 80);
   }
```

For more information, refer to "Compiler Built-In Functions" on page 1-126).

Smaller Applications: Optimizing for Code Size

The same philosophy for producing fast code also applies to producing small code. You should present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, and hence the greatest freedom to safely manipulate the code to produce small applications.

Once the program is presented in this way, the optimization strategy depends on the code-size constraint that the program must obey. The first step should be to optimize the application for full performance, using `-O` or `-ipa` switches. If this obeys the code-size constraints, then no more need be done.

The “optimize for space” switch `-Os` (on page 1-50). which may be used in conjunction with IPA, performs every performance-enhancing transformation except those that increase code size. In addition, the `-e` linker switch (`-flags-link -e` if used from the compiler command line) may be helpful (on page 1-35). This operation performs section elimination in the linker to remove unneeded data and code. If the code produced with `-Os` and `-e` does not meet the code-size constraint, some analysis of the source code is required to try to reduce the code size further.

Note that loop transformations such as unrolling and software pipelining increase code size. But these loop transformations also give the greatest performance benefit. Therefore, in many cases compiling for minimum code size produces significantly slower code than optimizing for speed.

The compiler provides a way to balance between the two extremes of `-O` and `-Os`. This is the sliding-scale `-Ov num` switch (adjustable using the optimization slider bar under **Project Options** in the VisualDSP++ IDDE), described on page 1-50. The `num` parameter is a value between 0 and 100, where the lower value corresponds to minimum code size and the upper to maximum performance. A value in-between is used to optimize the frequently-executed regions of code for maximum performance,

while keeping the infrequently-executed parts as small as possible. The switch is most reliable when using profile-guided optimization (see "Optimization Control" on page 1-81) since the execution counts of the various code regions have been measured experimentally. Without PGO, the execution counts are estimated, based on the depth of loop nesting.

Avoid the use of inline code.

Avoid using the `inline` keyword to inline code for functions that are used a number of times, especially if they not very small functions. The `-Os` switch does not have any effect on the use of the `inline` keyword. It does, however, prevent automatic inlining (using the `-Oa` switch) from increasing the code size. Macro functions can also cause code expansion and should be used with care.

Using Pragmas for Optimization

Pragmas can assist optimization by allowing the programmer to make assertions or suggestions to the compiler. This section looks at how they can be used to finely tune source code. Refer to "Pragmas" on page 1-179 for full details of how each pragma works. The emphasis of this section is to consider under what circumstances they are useful during the optimization process.

In most cases, the pragmas serve to give the compiler information that it is unable to deduce for itself. The programmer is responsible for making sure that the information given by the pragma is valid in the context in which it is used. If the programmer uses a pragma to assert that a function or loop has a quality it does not have may result in a malfunctioning application.

Pragmas are advantageous because they allow code to remain portable, since they normally are ignored by a compiler that does not recognize them.

Function Pragmas

Function pragmas include `#pragma const`, `#pragma pure`, `#pragma regs_clobbered`, and `#pragma optimize_{off|for_speed|for_space|as_cmd_line}`.

#pragma const

This pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers), and the result returned is only a function of the parameter values. The pragma may be applied to a function prototype or definition. It helps the compiler since two calls to the function with identical parameters always yield the same result. In this way calls to `#pragma const` functions may be hoisted out of loops if their parameters are loop independent.

#pragma pure

Like `#pragma const`, this pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers). However, the result returned may be a function of both the parameter values and any global variables. The pragma may be applied to a function prototype or definition. Two calls to the function with identical parameters always yield the same result, provided that no global variables have been modified between the calls. Hence, calls to `#pragma pure` functions may be hoisted out of loops if their parameters are loop independent and no global variables are modified in the loop.

#pragma regs_clobbered

This pragma is a useful way to improve the performance of code that makes function calls. The best use of the pragma is to increase the number of call-preserved registers available across a function call. There are two complementary ways in which this may be done.

First, suppose that you have a function written in assembly that you wish to call from C source code. The `regs_clobbered` pragma may be applied to the function prototype to specify which registers are "clobbered" by the assembly function, that is, which registers may have different values before and after the function call. Consider a simple assembly function that adds two integers, then masks the result to fit into 8 bits:

```
_add_mask:
   J8 = J4 + J5;;
   J0 = 255;;
   J8 = J8 AND J0;;
   cjmp (ABS);;
._add_mask.end:
```

The function does not modify the majority of the scratch registers available and thus these could instead be used as call-preserved registers. In this way, fewer spills to the stack would be needed in the caller function. Using the following prototype,

```
// GOOD: uses regs_clobbered to increase call-preserved register
set   #pragma regs_clobbered "J0, J8, jICC"
   #pragma regs_clobbered "J0, J8, jICC"
   int add_mask(int, int);
```

the compiler is told which registers are modified by a call to the `add_mask` function. The registers not specified by the pragma are assumed to preserve their values across such a call and the compiler may use these spare registers to its advantage when optimizing the call sites.

The pragma is also powerful when all of the source code is written in C. In the above example, a C implementation might be:

```
// BAD: function thought to clobber entire volatile register set
   int add_mask(int a, int b) {
      return ((a+b)&255);
   }
```

Since this function does not need many registers when compiled, it can be defined using

```
// GOOD: function compiled to preserve most registers
   #pragma regs_clobbered "J0, J8, CCset"
   int add_mask(int a, int b) {
       return ((a+b)&255);
   }
```

to ensure that any other registers aside from `J0, J8` and the condition codes are not modified by the function. If any other registers are used in the compilation of the function, they are saved and restored during the function prologue and epilogue.

In general, it is not very helpful to specify any of the condition codes as call-preserved as they are difficult to save and restore and are usually clobbered by any function. Moreover, it is usually of limited benefit to be able to keep them live across a function call. Therefore, it is better to use `CCset` (all condition codes) rather than `jICC` in the clobbered set above. For more information, refer to "#pragma regs_clobbered string" on page 1-195.

#pragma optimize_{off|for_speed|for_space}

This pragma may be used to change the optimization setting on a function-by-function basis. In particular, it may be useful to optimize functions that are rarely called (for example, error handling code) for space (`#pragma optimize_for_space`), whereas functions critical to performance should be compiled for maximum speed (using `#pragma optimize_for_speed`). The `#pragma optimize_off` is useful for debugging specific functions without increasing the size or decreasing the performance of the overall application unnecessarily.

The `#pragma optimize_as_cmd_line` resets the optimization settings to those specified on the `ccts` command line when the compiler is invoked. Refer to "General Optimization Pragmas" on page 1-203 for more information.

Loop Optimization Pragmas

Many pragmas are targeted towards helping to produce optimal code for inner loops. These are the `loop_count`, `no_vectorization`, `vector_for`, `all_aligned`, `different_banks`, and `no_alias` pragmas.

#pragma loop_count

This pragma enables the programmer to inform the compiler about a loop's iteration count. The compiler is able to make more reliable decisions about the optimization strategy for a loop if it knows the iteration count range. If you know that the loop count is always a multiple of some

constant, this can also be useful as it allows a loop to be partially unrolled or vectorized without the need for conditionally-executed iterations. Knowledge of the minimum trip count may allow the compiler to omit the guards that are usually required after software pipelining. Any of the parameters of the pragma that are unknown may be left blank.

The following is an example of the `loop_count` pragma:

```
// GOOD: the loop_count pragma gives compiler helpful information
to assist optimization
   #pragma loop_count(/*minimum*/ 40, /*maximum*/ 100, /*modulo*/ 4)
   for (i=0; i<n; i++)
       a[i] = b[i];
```

For more information, refer to "#pragma loop_count(min, max, modulo)" on page 1-188.

#pragma no_vectorization

Vectorization (executing more than one iteration of a loop in parallel) can slow down loops with very small iteration counts, since a loop prologue and epilogue are required. The `no_vectorization` pragma can be used directly above a `for` or `do` loop to tell the compiler *not* to vectorize the loop.

#pragma vector_for

This pragma is used to help the compiler to resolve dependencies that would normally prevent it from vectorizing a loop. It tells the compiler that all iterations of the loop may be run in parallel with each other, subject to rearrangement of reduction expressions in the loop. In other words, there are no loop-carried dependencies except reductions. An optional parameter, `n`, may be given in parentheses to say that only `n` iterations of the loop may be run in parallel. The parameter must be a literal value.

For example,

```
// BAD: cannot be vectorized due to possible alias between a and
b
   for (i=0; i<100; i++)
      a[i] = b[i] + a[i-4];
```

cannot be vectorized if the compiler cannot tell that the array `b` does not alias array `a`. But the pragma may be added to instruct the compiler that in this case four iterations may be executed concurrently.

```
// GOOD: pragma vector_for disambiguates alias
   #pragma vector_for (4)
   for (i=0; i<100; i++)
      a[i] = b[i] + a[i-4];
```

Note that this pragma does not force the compiler to vectorize the loop. The optimizer checks various properties of the loop and does not vectorize it if it believes that it is unsafe or cannot deduce information necessary to carry out the vectorization transformation. The pragma assures the compiler that there are no loop-carried dependencies, but there may be other properties of the loop that prevent vectorization.

In cases where vectorization is impossible, the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

For more information, refer to "#pragma vector_for" on page 1-192.

#pragma all_aligned

This pragma is used as shorthand for multiple `__builtin_aligned` assertions. By prefixing a `for` loop with the pragma, it is asserted that every pointer variable in the loop is aligned on a quad-word boundary at the beginning of the first iteration.

Therefore, adding the pragma to the following loop

```
// GOOD: uses all_aligned to inform compiler of alignment of a
and b
   #pragma all_aligned
   for (i=0; i<100; i++)
      a[i] = b[i];
```

is equivalent to writing

```
// GOOD: uses __builtin_aligned to give alignment of a and b
   __builtin_aligned(a, (sharc)/4/);
   __builtin_aligned(b, (sharc)/4/);
   for (i=0; i<100; i++)
      a[i] = b[i];
```

In addition, the `all_aligned` pragma may take an optional literal integer argument `n` in parentheses. This tells the compiler that all pointer variables are aligned on a quad-word boundary at the beginning of the n^{th} iteration. Note that the iteration count begins at zero.

Therefore,

```
// GOOD: uses all_aligned to inform compiler of alignment of a
and b
   #pragma all_aligned (3)
   for (i=99; i>=0; i--)
      a[i] = b[i];
```

is equivalent to

```
// GOOD: uses __builtin_aligned to give alignment of a and b
   __builtin_aligned(b+96, 4);
   __builtin_aligned(a+96, 4);
   __builtin_aligned(b+96, 4);
   for (i=99; i>=0; i--)
      a[i] = b[i];
```

For more information, refer to “Using __builtin_aligned” on page 2-18.

#pragma different_banks

This pragma is used as shorthand for declaring multiple pointer types with different bank qualifiers. It asserts that any two independent memory accesses in the loop may be issued together without incurring a stall.

For example,

```
// GOOD: uses different banks to allow simultaneous accesses to a
and b
   #pragma different_banks
   for (i=0; i<100; i++)
      a[i] = b[i] + a[i-4];
```

allows a single instruction loop to be created. Note that the use of the `different_banks` pragma implies that the memory accesses do not alias each other. If they did, it would not be permitted to issue them simultaneously. See "#pragma different_banks" on page 1-188 for more information.

#pragma no_alias

When immediately preceding a loop, the `no_alias` pragma asserts that no load or store in the loop accesses the same memory. This helps to produce shorter loop kernels because it permits instructions in the loop to be rearranged more freely. See "#pragma no_alias" on page 1-191 for more information.

Useful Optimization Switches

Table 2-2 lists the compiler switches useful during the optimization process.

Table 2-2. C/C++ Compiler Optimization Switches

Switch Name	Description
`-const-red-write` on page 1-29	Specifies that data accessed via a pointer to `const` data may be modified elsewhere
`-flags-link -e` on page 1-35	Specifies linker section elimination
`-force-circbuf` on page 1-35	Treats array references of the form `array[i%n]` as circular buffer operations
`-ipa` on page 1-40	Turns on inter-procedural optimization. Implies use of `-O`. May be used in conjunction with `-Os` or `-Ov`.
`-no-fp-associative` on page 1-46	Does not treat floating-point multiply and addition as an associative
`-no-saturation` on page 1-47	Does not turn non-saturating operations into saturating ones
`-O` on page 1-48	Enables code optimizations and optimizes the file for speed
`-Os` on page 1-50	Optimizes the file for size
`-Ov num` on page 1-50	Controls speed vs. size optimizations (sliding scale)
`-pguide` on page 1-56	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization
`-save-temps` on page 1-59	Saves intermediate files (for example, `.s`)

`-flags-link -e` on page 1-35	Specifies linker section elimination

How Loop Optimization Works

Loop optimization is important to overall application performance, because any performance gain achieved within the body of a loop reaps a benefit for every iteration of that loop. This section provides an introduction to some of the concepts used in loop optimization, helping you to use the compiler features in this chapter.

Terminology

This section describes terms that have particular meanings for compiler behavior.

Clobbered

A register is “clobbered” if its value is changed so that the compiler cannot usefully make assumptions about its new contents.

For example, when the compiler generates a call to an external function, the compiler considers all caller-preserved registers to be clobbered by the called function. Once the called function returns, the compiler cannot make any assumptions about the values of those registers. This is why they are called “caller-preserved.” If the caller needs the values in those registers, the caller must preserve them itself.

The set of registers clobbered by a function can be changed using `#pragma regs_clobbered`, and the set of registers changed by a gnu `asm` statement is determined by the clobber part of the `asm` statement.

Live

A register is “live” if it contains a value needed by the compiler, and thus cannot be overwritten by a new assignment to that register. For example, to do "A = B + C", the compiler might produce:

```
reg1 = load B      // reg1 becomes live
reg2 = load C      // reg2 becomes live
reg1 = reg1 + reg2 // reg2 ceases to be live;
                   // reg1 still live, but with a different
                   // value
store reg1 to A    // reg1 ceases to be live
```

Liveness determines which registers the compiler may use. In this example, since `reg1` is used to load `B`, and that value cannot be used until the addition, `reg1` cannot also be used to load the value of `C`, unless the value in `reg1` is first stored elsewhere.

Spill

When a compiler needs to store a value in a register, and all usable registers are already live, the compiler must store the value of one of the registers to temporary storage (the stack). This “spilling” process prevents the loss of a necessary value.

Scheduling

“Scheduling” is the process of determining a valid ordering –a schedule–of the program instructions. The compiler attempts to produce the most efficient schedule.

Loop kernel

The “loop kernel”is the body of code that is executed once per iteration of the loop. It excludes any code required to set up the loop or to finalize it after completion.

Loop prolog

A “loop prolog” is a sequence of code required to set the machine into a state whereby the loop kernel can execute. For example, the prolog usually involves ensuring that values are loaded into registers ready for use. Not all loops need a prolog.

Loop epilog

A “loop epilog” is a sequence of code responsible for finalizing the execution of a loop. After each iteration of the loop kernel, the machine will be in a state where the next iteration can begin efficiently. The epilog moves values from the final iteration to where they need to be for the rest of the function to execute. For example, the epilog might save values to memory. Not all loops need an epilog.

Loop invariant

A “loop invariant” is an expression that has the same value for all iterations of a loop. For example:

```
int i, n = 10;
for (i = 0; i < n; i++) {
    val += i;
}
```

The variable `n` is a loop invariant, as its value is not changed during the body of the loop, so `n` will have the value 10 for every iteration of the loop.

Hoisting

When the optimizer determines that some part of a loop is computing a value that is actually a loop invariant, it may move that computation to before the loop. This “hoisting” prevents the same value from being re-computed for every iteration.

Sinking

When the optimizer determines that some part of a loop is computing a value that is not used until the loop terminates, the compiler may move that computation to after the loop. This “sinking” process ensures the value is only computed using the values from the final iteration. Sinking prevents the compiler from repeatedly computing a value and then discarding all but the last value, unused.

Loop Optimization Concepts

The compiler optimizer focuses considerable attention on program loops, as any gain in the loop's performance reaps the benefits on every iteration of the loop. The applied transformations can produce code that appears to be substantially different from the structure of the original source code. This section provides a simple introduction to the compiler's loop optimization, to help you understand why the code might be different.

The following examples are presented in terms of a hypothetical machine. This machine is capable of issuing up to two instructions in parallel, provided one instruction is an arithmetic instruction, and the other is a load or a store. Two arithmetic instructions may not be issued at once, nor may two memory accesses:

```
t0 = t0 + t1;            // valid: single arithmetic
t2 = [p0];               // valid: single memory access
[p1] = t2;               // valid: single memory access
t2 = t3 + 4, t2 = [p2]; // valid: arithmetic and memory
t5 += 1, t6 -= 1;        // invalid: two arithmetic
[p3] = t2, t4 = [p5];    // invalid: two memory
```

The machine can use the old value of a register and assign a new value to it in the same cycle, for example:

```
t2 = t3 + 4, t2 = [p2]; // valid: arithmetic and memory
```

The value of `t1` on entry to the instruction is the value used in the addition. On completion of the instruction, `t1` contains the value loaded via the `p0` register.

The examples will show "`START LOOP N`" and "`END LOOP`", to indicate the boundaries of a loop that iterates `N` times. (The mechanisms of the loop entry and exit are not relevant).

Software pipelining

"Software pipelining" is analogous to hardware pipelining used in some processors. Whereas hardware pipelining allows a processor to start processing one instruction before the preceding instruction has completed, software pipelining allows the generated code to begin processing the next iteration of the original source-code loop before the preceding iteration is complete.

Software pipelining makes use of a processor's ability to multi-issue instructions. Regarding known delays between instructions, it also schedules instructions from later iterations where there is spare capacity.

Loop Rotation

"Loop rotation" is a common technique of achieving software pipelining. It changes the logical start and end positions of the loop within the overall instruction sequence, to allow a better schedule within the loop itself. For example, this loop:

```
START LOOP N
A
B
C
```

```
D
E
END LOOP
```

could be rotated to produce the following loop:

```
A
B
C
START LOOP N-1
D
E
A
B
C
END LOOP
D
E
```

The order of instructions in the loop kernel is now different. It still circles from instruction `E` back to instruction `A`, but now it starts at `D`, rather than `A`. The loop also has a prolog and epilog added, to preserve the intended order of instructions. Since the combined prolog and epilog make up a complete iteration of the loop, the kernel is now executing `N-1` iterations, instead of `N`.

In this example consider the following loop:

```
START LOOP N
t0 += 1
[p0++] = t0
END LOOP
```

This loop has a two-cycle kernel. While the machine could execute the two instructions in a single cycle – an arithmetic instruction and a memory access instruction – to do so would be invalid, because the second instruction depends upon the value computed in the first instruction. However, if the loop is rotated, we get:

```
t0 += 1
START LOOP N-1
[p0++] = t0
t0 += 1
END LOOP
[p0++] = t0
```

The value being stored is computed in the previous iteration (or before the loop starts, in the prolog). This allows the two instructions to be executed in a single cycle:

```
t0 += 1
START LOOP N-1
[p0++] = t0, t0 += 1
END LOOP
[p0++] = t0
```

Rotating the loop has presented an opportunity by which the k^{th} iteration of the original loop is starting (`t0 += 1`) while the $(k-1)^{th}$ iteration is completing (`[p0++] = t0`). As a result, rotation has achieved software pipelining, and the performance of the loop is doubled.

Notice that this process has changed the structure of the program slightly. Suppose that the loop construct always executes the loop at least once; that is, it is a `1..N` count. Then if `N==1`, changing the loop to be `N-1` would be problematic. In this example, the compiler inserts a conditonal jump around the loop construct for the circumstances where the compiler cannot guarantee that `N > 1`:

```
t0 += 1
IF N == 1 JUMP L1;
```

```
START LOOP N-1
[p0++] = t0, t0 += 1
END LOOP
L1:
[p0++] = t0
```

Loop Vectorization

"Loop vectorization" is another transformation that allows the generated code to execute more than one iteration in parallel. However, vectorization is different from software pipelining. Where software pipelining uses a different ordering of instructions to get better performance, vectorization uses a different set of instructions. This different set is chosen because the new instructions perform multiple versions of the individual operations in the corresponding original instruction.

For example, consider this dot-product loop:

```
int i, sum = 0;
for (i = 0; i < n; i++) {
    sum += x[i] * y[i];
}
```

This loop walks two arrays, reading consecutive values from each, multiplying them and adding the result to the on-going sum. This loop has these important characteristics:

- Successive iterations of the loop read from adjacent locations in the arrays.
- The dependency between successive iterations is the summation, a commutative operation.
- Operations such as load, multiply and add are often available in parallel versions on embedded processors.

These characteristics allow the optimizer to vectorize the loop so that two elements are read from each array per load, two multiplies are done, and two totals maintained. The vectorized loop would be:

```
t0 = t1 = 0
START LOOP N/2
t2 = [p0++] (Wide)  // load x[i] and x[i+1]
t3 = [p1++] (Wide)  // load y[i] and y[i+1]
t0 += t2 * t3 (Low), t1 += t2 * t3 (High) // vector mulacc
END LOOP
t0 = t0 + t1        // combine totals for low and high
```

Vectorization is possible only when all the operations in the loop can be expressed in terms of parallel operations. Loops with conditional constructs in them are rarely vectorizable, because the compiler cannot guarantee that the condition will evaluate in the same way for all the iterations being executed in parallel.

Vectorization is possible only when all the operations in the loop can be expressed in terms of parallel operations. Loops with conditional constructs in them are rarely vectorizable, because the compiler cannot guarantee that the condition will evaluate in the same way for all the iterations being executed in parallel.

Vectorization is also affected by data alignment constraints and data access patterns. Data alignment affects vectorization because processors often constrain loads and stores to be aligned on certain boundaries. While the unvectorized version will guarantee this, the vectorized version imposes a greater constraint that may not be guaranteed. Data access patterns affect vectorization because memory accesses must be contiguous. If a loop accessed every tenth element, for example, then the compiler would not be able to combine the two loads for successive iterations into a single access.

Vectorization divides the generated iteration count by the number of iterations being processed in parallel. If the trip count of the original loop is unknown, the compiler will have to conditionally execute some iterations of the loop.

If the compiler cannot determine whether the loop is vectorizable at compile-time and the speed/space optimization settings allow it, the compiler will generate vectorized and non-vectorized versions of the loop. It will select between the two at run-time. This allows for considerable performance improvements, at the expense of code-size and an initial set-up cost.

Vectorization and software pipelining are not mutually exclusive: the compiler may vectorize a loop and then use software pipeling to obtain better performance.

Modulo Scheduling

Loop rotation, as described earlier, is a simple software-pipelining method that can often improve loop performance, but more complex examples require a more advanced approach. The compiler uses a popular technique known as "Modulo Scheduling" which can produce more efficient schedules for loops than simple loop rotation.

Modulo Scheduling has some specific terms that are used to characterize the loops it optimizes; understanding these terms will allow you to interpret the generated code more easily. The compiler's annotations use these terms, so you can examine the source code and the generated instructions, and see how the scheduling relates to the original source. See “Assembly Optimizer Annotations” on page 2-72 for more information.

Modulo scheduling performs software pipelining by determining how many iterations of the loop can be executing in parallel, and how soon after the current iteration the next iteration can start. The delay between

the start of one iteration and the start of the next is known as the "iteration interval" ("II"). If the compiler can reduce the value for II, it can start the next iteration sooner, and thus increase the performance of the loop.

The II is limited by a number of factors, including:

- The number of instructions in the loop kernel
- Data dependencies on preceding instructions within the loop kernel
- The machine resources
- Stalls caused by particular instructions

For the purposes of this discussion, all instructions will be assumed to require only a single cycle to execute; on a real processor, stalls affect the Iteration Interval, so a loop that executes in II cycles may have fewer than II instructions.

For the preceding example, the Iteration Interval has the value II=1, because iteration *i*+1 can start on the cycle immediately after the cycle on which iteration *i* starts. See Table 2-3.

Table 2-3. Iteration Interval Cycles

Cycle	Iteration 1	Iteration 2	Iteration 3	Iteration 4
0	`t0 += 1`			
1	`[p0++] = t0`	`t0 += 1`		
2		`[p0++] = t0`	`t0 += 1`	
3			`[p0++] = t0`	`t0 += 1`
4				`[p0++] = t0`

Notice that cycles 1, 2 and 3 are each executing the same combination of instructions: `[p0++] = t0, t0 += 1`. This is the optimized loop kernel, and is also exactly the same as the rotated, software-pipelined loop given previously.

The iteration interval of the loop indicates several important characteristics of the loop:

- The loop kernel will be II cycles in length.
- A new iteration of the original loop will start every II cycles.
- The same instruction will execute on cycle c and on cycle c+II.

Variable expansion

Sometimes, the preferred schedule introduces some problems. For example, consider the following loop:

```
START LOOP N
t1 = [p1++]
t2 = [p2++]
t3 += t1*t2
t4 -= t1*t2
END LOOP
```

The instructions in this loop fit well with the machine's resources, in that the instructions parallelize well. Table 2-4 shows the iteration interval cycles:

Table 2-4. Iteration Interval Cycles with Variable Expansion

Cycle	Iteration 1	Iteration 2	Iteration 3	Iteration 4
0	`t1 = [p1++]`			
1	`t2 = [p2++]`			
2	`t3 += t1*t2`	`t1 = [p1++]`		

Table 2-4. Iteration Interval Cycles with Variable Expansion (Cont'd)

Cycle	Iteration 1	Iteration 2	Iteration 3	Iteration 4
3	`t4 -= t1*t2`	`t2 = [p2++]`		
4		`t3 += t1*t2`	`t1 = [p1++]`	
5		`t4 -= t1*t2`	`t2 = [p2++]`	
6			`t3 += t1*t2`	`t1 = [p1++]`
7			`t4 -= t1*t2`	`t2 = [p2++]`
8				`t3 += t1*t2`
9				`t4 -= t1*t2`

This example shows an iteration interval of II=2, as cycles 2, 4 and 6 are the start of repeating sequences. This would imply a loop of:

```
t1 = [p1++]
t2 = [p2++]
START LOOP N-1
t3 += t1*t2, t1 = [p1++]
t4 -= t1*t2, t2 = [p2++]
END LOOP
t3 += t1*t2
t4 -= t1*t2
```

However, there is a problem within the new loop kernel: `t1` is loaded on the first cycle and then used on the second cycle; however, the intended use was of the value loaded three cycles earlier, during the previous iteration. Thus, if the compiler were to use this schedule as-is, it would be clobbering the live value in `t1`. The lifetime of each value loaded into t1 is 4 cycles, but the loop's iteration interval is only 2, so the lifetimes of `t1` overlap.

The compiler fixes this by duplicating the kernel, so that:

```
START LOOP N-1
t3 += t1*t2, t1 = [p1++]
t4 -= t1*t2, t2 = [p2++]
END LOOP
```

becomes:

```
START LOOP (N-1)/2
t3 += t1*t2,   t1' = [p1++]
t4 -= t1*t2,   t2' = [p2++]
t3 += t1'*t2', t1  = [p1++]
t4 -= t1'*t2', t2  = [p2++]
END LOOP
```

This means that the length of the loop kernel itself is 4, matching the lifetimes of the values in the loop. Then the compiler renames the variables that clash–in this case, just `t1`:

```
START LOOP (N-1)/2
t3 += t1*t2,   t1_2 = [p1++]
t4 -= t1*t2,   t2 = [p2++]
t3 += t1_2*t2, t1  = [p1++]
t4 -= t1_1*t2, t2  = [p2++]
END LOOP
```

Now, there are two variables, `t1` and `t1_2`, each with a lifetime of 4 that is satisfied by the length of the kernel.

So the loop becomes:

```
t1 = [p1++]
t2 = [p2++]
START LOOP (N-1)/2
t3 += t1*t2,   t1_2 = [p1++]
t4 -= t1*t2,   t2 = [p2++]
```

```
t3 += t1_2*t2, t1  = [p1++]
t4 -= t1_1*t2, t2  = [p2++]
END LOOP
t3 += t1*t2
t4 -= t1*t2
```

This process of duplicating the kernel and renaming colliding variables is called "variable expansion," and the number of times the compiler has to do this is referred to as the "modulo variable exansion factor" (MVE). In terms of reading the code, this means that a single iteration of the loop generated by the compiler will be processing more than one iteration of the original loop. Also, the compiler will be using several sets of registers to allow the iterations of the original loop to overlap without clobbering the live values.

Notice that as the modulo scheduler expands the loop kernel to add in the extra variable sets, the iteration count of the generated loop changes from `(N-1)` to `(N-1)/2`–or to whatever value is required. This is because each iteration of the generated loop is now doing more than one iteration of the original loop, so fewer generated iterations are required. However, this also relies on the compiler knowing that it can divide the loop count in this manner. For example, if the compiler produces a loop with MVE=2 so that the count should be `(N-1)/2`, an odd value of `(N-1)` causes problems. In these cases, the compiler generates additional "peeled" iterations of the original to handle the remaining iteration. As with rotation, if the compiler cannot determine the value of `N`, it will make parts of the loop–the kernel or peeled iterations–conditional so that they are executed only for the appropriate values of `N`.

A Worked Example

The following floating-point scalar product loop is used to show how the optimizer works.

Example: C source code for floating-point scalar product.

```
float sp(float *a, float *b, int n) {
   int i;
   float sum=0;
   for (i=0; i<n; i++) {
      sum+=a[i]*b[i];
   }
   return sum;
}
```

After code generation and conventional scalar optimizations, the compiler generates a loop that resembles the following example.

Example: Initial code generated for floating-point scalar product

```
.P1L3:
    xR0 = [J0 += 1];;
    xR2 = [J1 += 1];;
    xfR4 = R0 * R2;;
    xfR5 = R5 + R4;;
    if nlce0, jump
.P1L3;;
```

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero. This enables the generation of a hardware loop. (`LC0` is initialized with the loop count.) The sum is being accumulated in `xR5`. `J0` and `J1` hold pointers that are initialized with the parameters `a` and `b` and are incremented on each iteration.

To use both compute blocks, the optimizer unrolls the loop to run two iterations in parallel. `sum` is now being accumulated in `xR5` and `yR5`, both of which must be added together after the loop to produce the final result. To use the long-word loads needed for the loop to be as efficient, the compiler has to know that `J0` and `J1` have initial values that are even. Note also that unless the compiler knows that the original loop was executed an even number of times, a conditionally-executed odd iteration must be inserted outside the loop. The initial value of `LC0` has been halved.

Example: Code generated for floating-point scalar product after vectorization transformation

```
.P1L3:
    yxR0 = l[J0 += 2];;
    yxR2 = l[J1 += 2];;
    xyfR4 = R0 * R2;;
    xyfR5 = R5 + R4;;
    if nlce0, jump .P1L3;;
```

If the optimizer can verify that `J0` and `J1` are initially quad-word-aligned, it unrolls the loop to make better use of the TigerSHARC processor memory bandwidth.

In the next step the `sum` is being calculated in `xR5`, `yR5`, `xR7` and `yR7`. `LC0` has been halved again.

Example: Code generated for floating-point scalar product after a second vectorization transformation

```
.P1L3:
    yxR1:0 = q[J0 += 4];;
    yxR3:2 = q[J1 += 4];;
    xyfR4 = R0 * R2;;
    xyfR5 = R5 + R4;;
    xyfR6 = R1 * R3;;
    xyfR7 = R7 + R6;;
    if nlce0, jump .P1L3;;
```

Finally, the optimizer software-pipelines the loop, unrolling and overlapping iterations to obtain highest use of functional units. The following code would be generated if it were known that the loop executed at least twenty times and that the loop count was a multiple of eight.

Example: Code generated for floating-point scalar product after software pipelining

```
.P1L3:
    yxR1:0 = q[J0+=4];;
    yxR3:2 = q[J1+=4];;
    yxR9:8 = q[J0+=4];;
    xyfR4 = R0 * R2; yxR11:10 = q[J1+=4];;
    xyfR6 = R1 * R3; yxR1:0 = q[J0+=4];;
    xyfR5 = R5 + R4; xyfR12 = R8 * R10; yxR3:2 = q[J1+=4];;

.P1L28:
    xyfR7 = R7 + R6; xyfR14 = R9 * R11; yxR9:8 = q[J0+=4];
    xyfR5 = R5 + R12; xyfR4 = R0 * R2; yxR11:10 = q[J1+=4];;
    xyfR7 = R7 + R14; xyfR6 = R1 * R3; yxR1:0 = q[J0+=4];;
    if nlce0, jump .P1L28; xyfR5 = R5 + R4; xyfR12 = R8 * R10;
                           yxR3:2 = q[J1+=4];;

    xyfR7 = R7 + R6; xyfR14 = R9 * R11;;
    xyfR5 = R5 + R12; xyfR4 = R0 * R2;;
    xyfR7 = R7 + R14; xyfR6 = R1 * R3;;
    xyfR5 = R5 + R4;;
    xyfR7 = R7 + R6;;
```

Assembly Optimizer Annotations

When the compiler optimizations are enabled, the compiler can perform a large number of optimizations to generate the resultant assembly code. The decisions taken by the compiler as to whether certain optimizations are safe or worthwhile are generally invisible to a programmer. However, it could be beneficial to get feedback from the compiler regarding the decisions made during optimization. The intention of the information

provided is to give a programmer an understanding of how close to optimal a program is and what more could possibly be done to improve the generated code.

The feedback from the compiler optimizer is provided by means of annotations made to the assembly file generated by the compiler. The assembly file generated by the compiler can be kept by specifying the `-S` switch (on page 1-59), the `-save-temps` switch (on page 1-59) or by checking the **Project Options->Compile->General->Save temporary files** option in VisualDSP++ IDDE.

There are several areas of information provided by the assembly annotations that could help code evaluation improve the generated code. For example, annotations could provide indications of resource usage or the absence of a particular optimization from the resultant code. Annotations of optimization absence can often be more important than those of its presence. Assembly code annotations give the programmer insight as to why the compiler enables and disables certain optimizations for a specific code sequence.

The assembly code generated by the compiler optimizer is annotated with the following information:

- “Procedure Statistics” on page 2-75
- “Loop Identification” on page 2-79
- “Vectorization” on page 2-85
- “Modulo Scheduling” on page 2-92

The assembly output for the examples in this chapter may differ based on optimization flags and the version of the compiler. As a result, you may not be able to reproduce these results exactly.

Global Information

For each compilation unit, the assembly output is annotated with the time of the compilation and the options used during that compilation.

For instance, if a file `hello.c` is compiled at 1pm, on December 7 using the command line:

```
ccts -O -S hello.c
```

then the `hello.s` file will show:

```
      .file “hello.s”

// compilation time: Wed Dec 07, 13:00:00 2005

// compilation options: -O -S
```

Procedure Statistics

For each function call, the following is reported:

- Frame size – The size of stack frame. In TigerSHARC processors, there are two stacks: `J` stack and `K` stack. Both are reported.
- Registers used – Since function calls tend to implicitly clobber registers, there are several sets:
    1. The first set is composed of the scratch registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.
    2. The second set are the call-preserved registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.
    3. The third set are the registers clobbered by the inner function calls.
- Inlined Functions – If inlining happens, then the header of the caller function reports which functions were inlined inside it and where. Each inlined function is reported using the position of the inlined call. All the functions inlined inside the inlined function are reported as well, generating in fact a tree of inlined calls. Each node, except the root, has the form:

  file_name:*line*:*column*'*function_name*

where:

- *function_name* is the name of the function inlined.
- *line* is the line number of the call to *function_name*, in the source file.
- *column* is the column number of the call to *function_name*, in the source file.
- *file_name* is the name of the source file calling *function_name*.

Example A (Procedure Statistics, for ADSP-TS101 Processors)

Consider the following program:

```
struct str {
    int x1, x2;
};

int func1(struct str*, int *);
int func2(struct str s);
int foo(int in)
{
    int sum = 0;
    int local;
    struct str l_str;
    sum += func1(&l_str, &local);
    sum += func2(l_str);
    return sum;
}
```

The procedure statistics for `foo` are:

```
_foo:
//-------------------------------------------------------------
//   Procedure statistics:
//
//   J Frame size            = 16 words
```

```
//   K Frame size           =  8 words
//
// Scratch registers modified:{XR4-XR5,J4-J6,J8,CJMP,jICC,kICC}
//
// Call preserved registers used:{J16-J19}
//
// Registers clobbered by function calls:
// {XR0-XR23,YR0-YR23,J0-J15,J26-J30,JB0-JB3,JL0-JL3,K0-K15,
// K26-K30,KB0-KB3,KL0-KL3,CJMP,LC0-LC1,XMR0-XMR3,YMR0-YMR3,
// XPR0-XPR1,YPR0-YPR1,XTR0-XTR15,YTR0-YTR15,XTHR0-XTHR1,
// YTHR0-YTHR1,jICC,kICC,XSTAT,YSTAT,XDAB0-XDAB3,YDAB0-YDAB3}
//-------------------------------------------------------------
// "ExampleA.c" line 6 col 5
   J26 = J27 - 64;K26 = K27 - 64;;
   [J27 += -16] = CJMP;q[K27 += -8] = J19:16;;
   // “ExampleA.c" line 11 col 5
   J5 = J26 + 63;;
   J4 = J26 + 61;;
   call _func1; q[J27+4]=J27:24; q[K27+4]=K27:24;;
// "ExampleA.c" line 12 col 5
   J6 = j31 + 1;;
   XR5 = [J26 + 62];;
   XR4 = [J26 + 61];;
   J16 = J8 + 0;;
   call _func2; q[J27+4]=J27:24; q[K27+4]=K27:24;;
   J8 = J8 + J16;;
   CJMP = [J26 + 64];J19:16 = q[K27 + 8];;
//        -- 11 stalls --
   cjmp (ABS); J27:24=q[J26+68]; K27:24=q[K26+68];;
      _foo.end:
      .global _foo;
      .type _foo,STT_FUNC;
```

Notes:

- The `J` Frame size is 16 words, and the `K` Frame size is 8 words, because this is the amount of stack space allocated by the function.
- The set of Scratch registers modified is `{XR4-XR5,J4-J6,J8,CJMP,jICC,kICC}` because except for the `func1` and `func2` function calls, these are the only scratch registers changed by `foo`.
- The set of Call preserved registers used is `{J16-J19}` because these are the only call preserved registers used by `foo`.
- The set of registers clobbered by function calls contains the set of registers potentially changed by the calls to `func1` and `func2`.

Example B (Inlining Summary)

This is an example of inlined function reporting.

```
 1 void f4(int n);
 2 __inline void f3(int n)
 3 {
 4   f4(n);
 5 }
 6
 7 __inline void f2(int n)
 8 {
 9   while (n--) {
10     f3(n);
11     f3(2*n);
12   }
13 }
14 void f1(volatile unsigned int i)
15 (
16   f2(30);
17 }
```

`f1` inlines the call of `f2`, which inlines the call of `f3` in two places. The procedure statistics for `f1` reports these inlined calls:

```
_f1:
//--------------------------------------------------------------
//  Procedure statistics
. . . . .
//Inlined in _f1:
//    ExampleB.c:16:7'_f2
//        ExampleB.c:11:11'_f3
//        ExampleB.c:10:11'_f3
//--------------------------------------------------------------
. . . . .
```

`f1` reports that `f2` was inlined at line 16 (column 7) and, implicitly, `f1` also inlined the two calls of `f3` inside `f2`.

Loop Identification

One useful annotation is loop identification—that is, showing the relationship between the source program loops and the generated assembly code. This is not easy due to the various loop optimizations. Some of the original loops may not be present, because they are unrolled. Other loops get merged, making it difficult to describe what has happened to them.

Finally, the assembly code may contain compiler-generated loops that do not correspond to any loop in the user program, but rather represent constructs such as structure assignment or calls to `memcpy`.

Loop Identification Annotations

Loop identification annotation rules are:

- Annotate only the loops that originate from the C looping constructs `do`, `while`, and `for`. Therefore, any `goto` defined loop is not accounted for.
- A loop is identified by the position of the corresponding keyword (`do`, `while`, `for`) in the source file.
- Account for all such loops in the original user program.
- Generally, loop bodies are delimited between the `Lx: Loop` at `<file position>` and `End Loop Lx` assembly annotation. The former annotation follows the label of the first block in the loop. The later annotation follows the first jump back to the beginning of the loop. However, there are cases in which the code corresponding to a user loop cannot be entirely represented between two markers. In such cases the assembly code contains blocks that belong to a loop, but are not contained between that loop's end markers. Such blocks are annotated with a comment identifying the innermost loop they belong to, `Part of Loop Lx`.
- Sometimes a loop in the original program does not show up in the assembly file because it was either transformed or deleted. In either case, a short description of what happened to the loop is given at the beginning of the function.

- A program's innermost loops are those loops that do not contain other loops. In addition to regular loop information, the innermost loops with no control flow and no function calls are annotated with additional information such as:
    - **Cycle count.** The number of cycles needed to execute one iteration of the loop, including the stalls.
    - **Resource usage.** The resources used during one iteration of the loop. For each resource we show how many of that resource are used, how many are available and the percentage of utilization during the entire loop. Resources are shown in decreasing order of utilization. Note that 100% utilization means that the corresponding resource is used at its full capacity and represents a bottleneck for the loop.
    - **Register usage.** If the compilation flag `-annotate-loop-instr` is used then the register usage table is shown. This table has one column for every register that is defined or used inside the loop. The header of the table shows the names of the registers, written on the vertical, top down. The registers that are not accessed do not show up. The columns are grouped on data registers, pointer registers and all other registers. For every cycle in a loop (including stalls) there is a row in the array. The entry for a register has a '*' on that row if the register is either live or being defined at that cycle.
    - **Optimizations.** Some loops are subject to optimizations such as vectorization. These loops receive additional annotations as described in the vectorization section.

Example C (Loop Identification, for ADSP-TS101 Processors)

Consider the following example:

```
1   int bar(int a[10000])
2   {
3      int i, sum = 0;
4      for (i = 0; i < 9999; ++i)
5          sum += (sum + 1);
6      while (i-- < 9999) /* this loop doesn't get executed */
7          a[i] = 2*i;
8      return sum;
9 }
```

The two loops are accounted for as follows:

```
_bar:
//--------------------------------------------------------------
..................  procedure statistics  .........
//--------------------------------------------------------------
// Original Loop at "ExampleC.c" line 6 col 5 -- loop structure
   removed due to constant propagation.
//--------------------------------------------------------------
//"ExampleC.c" line 1 col 5
    YR0 = 0;;
//"ExampleC.c" line 4 col 5
    LC0 = 9999;;
.P1L1:
//--------------------------------------------------------------
// Loop at "ExampleC.c" line 4 col 5
//--------------------------------------------------------------
// This loop executes 1 iteration of the original loop in 4 cycles.
// (cycle count 4 includes 2 stalls)
//--------------------------------------------------------------
// This loop's resource usage is:
//    Y Compute Block ALU  used   2 out of    4 ( 50.0%)
//    Y Compute Block      used   2 out of    8 ( 25.0%)
//--------------------------------------------------------------
//"ExampleC.c" line 5 col 2
//     -- stall --
       YR1 = INC R0;;
//     -- stall --
       if nlc0e, jump .P1L1 ;YR0 = R0 + R1;;
//--------------------------------------------------------------
```

```
// End Loop L1
//------------------------------------------------------------
//------------------------------------------------------------
// Part of top level (no loop)
//------------------------------------------------------------
//"ExampleC.c" line 8 col 5
    J8 = YR0;;
    cjmp (ABS);;

    _bar.end:
```

Notes:

- The keywords identifying the two loops are:

  1. `for` – Its position is in the file `ExampleC.c`, line 4, column 5.

  2. `while` – Its position is in the file `ExampleC.c`, line 6, column 5.

- Immediately after the procedure statistics, there is a message stating that the loop at line 6 in the user program was removed. The reason was constant propagation, which in this case realizes that the value of `I` after the first loop is 9999 and the second loop does not get executed.

- The start of the loop at line 4 is marked in the assembly by the '`Loop at ExampleC.c, line 4, column 5`' annotation. This annotation follows the loop label `.P1L1`. The loop label "`End Loop L1`" is used to identify the end of the loop.

- The loop resource information accounts for all instructions and stalls inside the loop. In our case, the loop body is executed in four cycles (2 stalls and 2 instruction lines). During these four cycles the Y ALU resource could have been used 4 times, but it is only used 2 times, yielding a 50% utilization. Note that the first stall is caused by the dependence between the write to `YR0` in the last line and the use of `YR0` in the next iteration of the loop.

File Position

As seen in Example C, a file position is given using the file name, line number and the column number in that file: `"ExampleC.c" line 4 col 5`.

This scheme uniquely identifies a source code position, unless inlining is involved. In presence of inlining, a piece of code from a certain file position can be inlined at several places, which in turn can be inlined at other places. Since inlining can happen for an unspecified number of times, a recursive scheme is used to describe a general file position.

Therefore, a `<general file position>` is `<file position>` inlined from `<general file position>`.

Example D (Inlining Locations)

Consider the following source code:

```
5   void f2(int n);
6   inline void f3(int n)
7   {
8      while(n--)
9           f4();
10         if (n == 7)
11            f2(3*n);
12  }
13
14  inline void f2(int n)
15  {
16     while(n--) {
17          f3(n);
18          f3(2*n);
19     }
20  }
21  void f1(volatile unsigned int i)
22  {
23     f2(30);
24  }
```

Here is some of the code generated for function `f1`:

```
_f1:
.P2L1:
//------------------------------------------------------------
// Loop at "ExampleD.c" line 16 col 5 inlined from
// "ExampleD.c" line 23 col 7
//------------------------------------------------------------
// "ExampleD.c" line 8 col 5
   YR0 = PASS R24; YR29 = YR27;;
   if yaeq, jump .P2L2 (NP);  else, YR30 = YR27;
   else, YR31 = LSHIFT R28 BY 0;;

.P2L4:
//------------------------------------------------------------
// Loop at "ExampleD.c" line 8 col 5 inlined from
// "ExampleD.c" line 17 col 4 inlined from "ExampleD.c"
// line 23 col 7
//------------------------------------------------------------
//------------------------------------------------------------
// End Loop L4
//------------------------------------------------------------

...

.P2L9:
//------------------------------------------------------------
// Loop at "ExampleD.c" line 8 col 5 inlined from "ExampleD.c"
// line 18 col 4 inlined from "ExampleD.c" line 23 col 7
//------------------------------------------------------------

//------------------------------------ --------------------
// End Loop L9
//------------------------------------------------------------
// End Loop L1
```

Vectorization

The trip count of a loop is the number of times the loop body gets executed.

Under certain conditions, the compiler can take two operations executed in consecutive iterations of a loop and to execute them in only one more powerful instruction. This gives a loop a smaller trip count. The transformation in which operations from two subsequent iterations are executed in one more powerful single operation is called “vectorization.”

For instance, the original loop may start with a trip count of 1000.

```
for(i=0; i< 1000; ++i)
   a[i] = b[i] + c[i];
```

and, after the optimization, end up with the vectorized loop with a final trip count of 500. The vectorization factor is the number of operations in the original loop that are executed at once in the transformed loop. It is illustrated using some pseudo code below.

```
for(i=0; i< 500; i+=2)
   (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
```

In the above example, the vectorization factor is 2. A loop may be vectorized more than once.

If the trip count is not a multiple of the vectorization factor, some iterations need to be peeled off and executed unvectorized. If in the previous example, the trip count of the original loop was 1001, then the vectorized code would be:

```
for(i=0; i< 500; i+=2)
   (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
a[1000] = b[1000] + c[1000];
// This is one iteration peeled from
// the back of the loop.
```

In the above examples, the trip count is known and the amount of peeling is also known. If the trip count (a variable) is not known, the number of peeled iterations depends on the trip count. In such cases, the optimized code contains peeled iterations that are executed conditionally.

Unroll and Jam

Another vectorization related transformation is unroll and jam. Let's consider the following function:

```
/* unroll and jam example */
void f_unroll_and_jam(int a[][40], int *restrict c) {
   int i, j;
   for (i=0; i<60; i++) {
       int sum=0;
       for (j=0; j<40; j++) {
            sum += a[j][i];
       }
       c[i] = sum;  }
}
```

The outer loop can be unrolled twice and the result is:

```
void f_unroll_and_jam(int a[][40], int *restrict c) {
   int i, j;
   for (i=0; i<30; i+=2) {
      {
         int sum=0;
         for (j=0; j<40; j++) {
             sum += a[j][i];
  }
         c[i] = sum;
      }
      {
         int sum=0;
         for (j=0; j<40; j++) {
             sum += a[j][i+1];
         }
         c[i+1] = sum;
      }
   }
}
```

The two inner loops can be jammed together. We shall assume that we have a `plus_eq2` operation which is a more powerful version of += that can handle two integers at a time. The result is:

```
void f_unroll_and_jam(int a[][40], int *restrict c) {
   int i, j;
   for (i=0; i<30; i+=2) {
       int sum0=0;
       int sum1=0;
       for (j=0; j<40; j++) {
           (sum0, sum1) .plus_eq2. (a[j][i], a[j][i+1]);
       }
       (c[i], c[i+1]) = (sum0, sum1);
   }
}
```

The above sequence of transformation, where an outer loop is unrolled and the copies corresponding to the inner loops are jammed together is called unroll and jam.

Example E (Unroll and Jam):

The assembly-annotated code for the above `f_unroll_and_jam` example is:

```
.P1L1:
//-------------------------------------------------------------
//  Loop at "ExampleE.c" line 3 col 3
//-------------------------------------------------------------
//  Loop was unrolled for unroll and jam 2 times
//-------------------------------------------------------------
.................. some outer loop code .........
.P1L10:
//-------------------------------------------------------------
//  Loop at "ExampleE.c" line 5 col 5
.................. other loop annotations .........
//-------------------------------------------------------------
//  Loop was jammed by unroll and jam 2 times
//-------------------------------------------------------------
.................. jammed loop code .........
//-------------------------------------------------------------
//  End Kernel for Loop L10
//-------------------------------------------------------------
```

```
.P1L11:
.................. more outer loop code .........
    if nlc0e, jump .P1L1 ; [J5 + -1] = YR0;;
//-------------------------------------------------------------
//  End Loop L1
//-------------------------------------------------------------
```

Loop Flattening

Another transformation, related to vectorization, is "loop flattening." Loop flattening takes two nested loops that run `N1` and `N2` times respectively, and transforms them into a single loop that runs `N1*N2` times.

Example F (Loop Flattening):

For instance, the following function

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   for (i=0; i< 30; ++i)
      for (j=0; j < 100; ++j)
          a[i][j] = b[i][j];
}
```

is transformed into

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   int *p_a = &a[0][0];
   int *p_b = &b[0][0];
   for (i=0; i< 3000; ++i)
       p_a[i] = p_b[i];
}
```

This may further facilitate the vectorization process:

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   int *p_a = &a[0][0];
   int *p_b = &b[0][0];
   for (i=0; i< 3000; i+=2)
```

```
        (p_a[i], p_a[i+1]) = (p_b[i], p_b[i+1]);
}
```

The assembly output for the loop flattening example is:

```
_copy_v:
//-------------------------------------------------------------
.................. procedure statistics .........
//------------------------------------------------------------ --
// Original Loop at "test_flatten_loop_dim.c" line 3 col 5 --
// loop flatten into Loop at "test_flatten_loop_dim.c" line 4 col 2
//-------------------------------------------------------------
.................. procedure code .........
.P1L1:
//-------------------------------------------------------------
// Loop at "test_flatten_loop_dim.c" line 4 col 2
//-------------------------------------------------------------
..................  loop annotations .........
..................  loop body .........
   if nlc0e, jump .P1L1 ; [J4 + -1] = XR3;;
//-------------------------------------------------------------
// End Loop L1
//-------------------------------------------------------------
```

Vectorization Annotations

For every loop that is vectorized, the following information is provided:

- The vectorization factor
- The number of peeled iterations
- The position of the peeled iterations (front or back of the loop)
- Information about whether peeled iterations are conditionally or unconditionally executed

For every loop pair that is subject to unroll and jam, it is necessary to:

- Annotate the unrolled outer loop with the number of times it was unrolled
- Annotate the inner loop with the number of times the loop was jammed

For every loop pair that is subject to loop flattening, it is necessary to account for the loop that is lost and show the remaining loop that it was merged with.

Example G (Vectorization, for ADSP-TS101 Processors):

Consider the test program:

```
void add(int *a, int *restrict b, int *restrict c, int dim) {
   int i;
   for (i = 0 ; i < dim; ++i)
       a[i] = b[i] + c[i];
}
```

for which the vectorization information is:

```
.P1L24:
//-------------------------------------------------------------
// Loop at "ExampleG.c" line 3 col 5
.................. other loop annotations .........
//-------------------------------------------------------------
// Loop was vectorized by a factor of 4.
//-------------------------------------------------------------
// Vectorization peeled 3 conditional iterations from the back
// of the loop because of an unknown trip count, possible
// not a multiple of 4.
//-------------------------------------------------------------
.................. loop body .........
   if nlc0e, jump .P1L24 ; [J4 + -1] = YR9 ; YR1:0 = XR3:2;;
//-------------------------------------------------------------
// End Kernel for Loop L24
//-------------------------------------------------------------
```

In this example, the vectorization factor is 4. Since the trip count `dim` is unknown, three conditional iterations are peeled from the back of the loop, corresponding to the three cases when `dim` is `4k+1`, `4k+2` or `4k+3`. The loop end is marked with `End Kernel`, because in this case the loop was modulo scheduled (see "Modulo Scheduling" on page 2-92).

Modulo Scheduling

"Modulo scheduling" is a kind of software pipelining. (See "Modulo Scheduling" on page 2-92.) It is used to schedule innermost loops without control flow. There are various algorithms for modulo scheduling, and all of them produce scheduled loops that can be described by the following parameters:

- Initiation interval (II): the number of cycles between initiating two successive iterations of the loop
- Stage count (SC): the number of initiation intervals until the first iteration of the loop has completed. This is also the number of iterations in progress at any time within the kernel.
- Modulo variable expansion unroll factor (MVE unroll): the number of times the loop has to be unrolled to achieve the schedule without overlapping register lifetimes.
- Trip count: the number of times the loop goes around
- Trip modulo: a number that is known to divide the trip count
- Trip maximum: an upper limit for the trip count
- Trip minimum: a lower limit for the trip count
- Minimum initiation interval due to resources (`res MII`): a lower limit for the Initiation Interval (II), imposed by the fact that at least one of the resources is used at maximum capacity.

The original loop instructions end up in three places: the prolog, the kernel and the epilog of the scheduled loop. The kernel is the most important and it is the body of the scheduled loop. The prolog and the epilog contain instructions peeled from the original loop that precede and follow the kernel, respectively.

Modulo Scheduling Annotations:

For every modulo scheduled loop, in addition to regular loop annotations, the following information is provided:

- The Initiation interval (II)
- The final trip count if it is known: the trip count of the loop as it ends up in the assembly code
- A cycle count representing the time to run one iteration of the pipelined loop
- The minimum trip count, if it is known and the trip count is unknown
- The maximum trip count, if it is known and the trip count is unknown
- The trip modulo, if it is known and the trip count is unknown
- The stage count (iterations in parallel)
- The MVE unroll factor
- The resource usage
- The minimum initiation interval due to resources (`res MII`)
- The schedule for one of the MVE iterations unrolled into the kernel (if MVE > 1)

The compiler's `-annotate-loop-instr` switch (on page 1-27) can be used to produce additional annotation information for the instructions that belong to the prolog, kernel or the epilog of the modulo scheduled loop. In this case, the instructions are annotated with the following information:

- The loop label the instruction is related to
- The part of the modulo scheduled loop (prolog, kernel or epilog) that the instruction belongs to
- ID: a unique number associated with the original instruction in the unscheduled loop that generates the current instruction. It is useful because a single instruction in the original loop can expand into multiple instructions in a modulo scheduled loop.
- Loop-carried dependencies bound and loop-carry path.

The IDs are assigned in the order the instructions appear in the kernel (though they might repeat for MVE unroll > 1).

- `sn`: the stage count the instruction belongs to. If the loop has a stage count of 1, the instruction's `sn` is not shown.
- `rs`: the register set used for the current instruction (useful when MVE unroll > 1, in which case `rs` can be `0,1,...mve-1`). If the loop has an MVE of 1, the instruction's `rs` is not shown.

- In addition to the above, the instructions in the kernel are annotated with:
  - `Iter`: specifies what iteration of the original loop an instruction is on in the schedule.
  - In a modulo scheduled kernel, there are instructions from (`SC+MVE-1`) iterations in the original loop. Each of them can be chosen as the current iterations, relative to which the other iterations are specified. Chose `Iter=0` for the instructions in the earliest iterations of the original loop.

If the annotations for modulo-scheduled instructions are enabled, then the format of an assembly line containing several instructions is changed from:

```
instruction_1; instruction_2; instruction_3;;
```

to:

```
/**/ instruction_1;
     instruction_2;
     instruction_3;;
```

This is done to allow for annotations at the instruction level:

```
/**/ instruction_1;   // {annotations for instruction_1}
     instruction_2;   // {annotations for instruction_2}
     instruction_3;;  // {annotations for instruction_3}
```

The `/**/` marks the beginning of an instruction line.

Example H (Modulo Scheduling):

Consider the following function:

```
1   void add(int *a, int *restrict b, int *restrict c) {
2      int i;
3      for (i = 0; i < 200; ++i)
```

```
4           a[i] = b[i] + c[i];
5   }
```

The annotated output for the modulo scheduling, including the optional annotation of the modulo-scheduling-related instructions, is:

```
8   _add:
9   //------------------------------------------------------------
10  // Procedure statistics:
11  //
12  //   J Frame size         = 12 words
13  //   K Frame size         =  4 words
14  //
15  // Scratch registers modified:
          {XR0-XR11,YR0-YR5,YR8-YR9,J0-J1,J4,
           JB0,JL0,LC0,xACC,yACC,jICC,kICC}
16  //
17  // No call preserved registers used.
18  //------------------------------------------------------------
19  // ExampleH.c" line 1 col 6
20          /**/ J26 = J27 - 64;
21               K26 = K27 - 64;;
22          /**/ J27 = J27 - 12;
23               K27 = K27 - 4;;
24          /**/ JB0 = 0;
25               J0 = J6 + 0;;
26          /**/ JL0 = 0;
27               J1 = J5 + 0;;
28          /**/ YR3:0 = DAB q[J1 += 4];
29               LC0 = 25;;
30          /**/ YR3:0 = DAB q[J1 += 4];;
                          // {L7 prolog:id=3,sn=0,rs=0}
31          /**/ XR3:2 = YR1:0;;   // {L7 prolog:id=6,sn=0,rs=0}
32  //        -- stall --
33          /**/ XR7:4 = DAB q[J0 += 4];;
34          /**/ XR7:4 = DAB q[J0 += 4];;
                          // {L7 prolog:id=1,sn=0,rs=0}
35          /**/ YR5:4 = XR7:6;;   // {L7 prolog:id=9,sn=0,rs=0}
36
37  .P1L7:
38  //------------------------------------------------------------
39  // Loop at "ExampleH.c" line 3 col 5
```

```
40  //
41  // This loop executes 8 iterations of the original loop
       in 12 cycles.
42  //------------------------------------------------------------
43  // Trip Count  = 24
44  //------------------------------------------------------------
45  // Successfully found modulo schedule with:
46  // Initiation Interval (II)         = 6
47  // Stage Count (SC)                 = 2
48  // MVE Unroll Factor                = 2
49  // Minimum initiation interval due to resources
       (res MII)                        = 6.00
50  //------------------------------------------------------------
51  // This loop's resource usage is:
52  //
53  //
54  //
55  //              resourse statistics
56  //
57  //
58  //
59  //
60  //
61  //------------------------------------------------------------
62  // Loop was vectorized by a factor of 4.
63  //------------------------------------------------------------
64          /**/ XR11:8 = DAB q[J0 += 4];;
                         // {L7 kernel:id=1,sn=0,rs=1,Iter=1}
65  // "ExampleH.c" line 4 col 2
66       /**/ xyR5:4 = R5:4 + R3:2;
                        // {L7 kernel:id=2,sn=1,rs=0,Iter=0}
67              YR3:0 = DAB q[J1 += 4];;
                        // {L7 kernel:id=3,sn=0,rs=1,Iter=1}
68       /**/ [J4 += 4] = XR4;;
                        // {L7 kernel:id=4,sn=1,rs=0,Iter=0}
69       /**/ [J4 + -3] = XR5;
                        // {L7 kernel:id=5,sn=1,rs=0,Iter=0}
70              XR3:2 = YR1:0;;
                        // {L7 kernel:id=6,sn=0,rs=1,Iter=1}
71       /**/ [J4 + -2] = YR4;;
                        // {L7 kernel:id=7,sn=1,rs=0,Iter=0}
```

```
72      /**/ if lc0e, jump .P1L8 (NP);
                        // {L7 kernel:id=10,sn=0,rs=1,Iter=1}
73            [J4 + -1] = YR5;
                        // {L7 kernel:id=8,sn=1,rs=0,Iter=0}
74            YR9:8 = XR11:10;;
                        // {L7 kernel:id=9,sn=0,rs=1,Iter=1}
75      /**/ XR7:4 = DAB q[J0 += 4];;
                        // {L7 kernel:id=1,sn=0,rs=0,Iter=2}
76      /**/ xyR9:8 = R9:8 + R3:2;
                        // {L7 kernel:id=2,sn=1,rs=1,Iter=1}
77            YR3:0 = DAB q[J1 += 4];;
                        // {L7 kernel:id=3,sn=0,rs=0,Iter=2}
78      /**/ [J4 += 4] = XR8;;
                        // {L7 kernel:id=4,sn=1,rs=1,Iter=1}
79      /**/ [J4 + -3] = XR9;
                        // {L7 kernel:id=5,sn=1,rs=1,Iter=1}
80            XR3:2 = YR1:0;;
                        // {L7 kernel:id=6,sn=0,rs=0,Iter=2}
81      /**/ [J4 + -2] = YR8;;
                        // {L7 kernel:id=7,sn=1,rs=1,Iter=1}
82      /**/ jump .P1L7 ;
83            [J4 + -1] = YR9;
                        // {L7 kernel:id=8,sn=1,rs=1,Iter=1}
84            YR5:4 = XR7:6;;
                        // {L7 kernel:id=9,sn=0,rs=0,Iter=2}
85  //-----------------------------------------------------------
86  // End Kernel for Loop L7
87  //-----------------------------------------------------------
88
89  .P1L8:
90      /**/ xyR1:0 = R9:8 + R3:2;;
                                  // {L7 epilog:id=2,sn=1,rs=1}
91      /**/ [J4 += 4] = XR0;;
                                  // {L7 epilog:id=4,sn=1,rs=1}
92      /**/ [J4 + -1] = YR1;;    // {L7 epilog:id=8,sn=1,rs=1}
93      /**/ [J4 + -2] = YR0;;    // {L7 epilog:id=7,sn=1,rs=1}
94      /**/ [J4 + -3] = XR1;;    // {L7 epilog:id=5,sn=1,rs=1}
95      /**/ cjmp (ABS); J27:24=q[J26+68]; K27:24=q[K26+68];;
96
97  ._add.end:
98        .global _add;
99        .type _add,STT_FUNC;
```

Notes:

Lines 43-60 define the kernel information: loop name and modulo schedule parameters: II, stage count, etc.

Lines 64-84 show the kernel.

Each instruction in the kernel has an annotation between {}, inside a comment following the instruction. If several instructions are executed in parallel, each gets its own annotation.

For instance, line 64 looks like:

```
64   /**/ XR11:8 = DAB q[J0 += 4];;
                // {L7 kernel:id=1,sn=0,rs=1,Iter=1}
```

This annotation describes:

- That this instruction belongs to the kernel of the loop starting at `L7`.
- `T`hat this and the other two instructions that have `ID=1` originate from the same original instruction in the unscheduled loop:

```
34      /**/ XR7:4 = DAB q[J0 += 4];;
                // {L7 prolog:id=1,sn=0,rs=0}
64      /**/ XR11:8 = DAB q[J0 += 4];;
                // {L7 kernel:id=1,sn=0,rs=1,Iter=1}
75      /**/ XR7:4 = DAB q[J0 += 4];;
                // {L7 kernel:id=1,sn=0,rs=0,Iter=2}
```

- `sn=0` shows that this instruction belongs to stage count 0.
- `rs=1` shows that this instruction uses register set 1.
- `Iter=1` specifies that this instruction belongs to the second iteration of the original loop (`Iter` numbers are zero-based).

The prolog and epilog are not clearly delimited in blocks by themselves, but their corresponding instructions are annotated similar to the ones in the kernel except that they do not have an `Iter` field and that they are preceded by a tag specifying whose loop prolog or epilog they belong to.

Examples:

```
30      /**/ YR3:0 = DAB q[J1 += 4];;
                 // {L7 prolog:id=3,sn=0,rs=0}
90      /**/ xyR1:0 = R9:8 + R3:2;;
                 // {L7 epilog:id=2,sn=1,rs=1}
```

Note that the prolog/epilog instructions may mix with other instructions on the same line.

This situation does not occur in this example; however, in a different example it might have:

```
        /**/ xyR0 = R0 + R3;    // {L10 epilog:id=2,sn=1,rs=1}
             J4 = J4 + 2;;
```

This shows a line with two instructions. The second instruction "`J4 = J4 + 2`" is unrelated to modulo scheduling, and therefore it has no annotation.

Warnings, Failure Messages and Advice

There are apparently innocuous programming constructs that have a negative effect on performance. Since you may not be aware of the hidden problems, the compiler annotations try to give warnings when such situations occur. Also, if a program construct keeps the compiler from performing a certain optimization, the compiler gives the reason why that optimization was precluded.

In some cases, the compiler thinks it could do a better job if you changed your code in certain ways. In these cases, the compiler offers advice on the potentially beneficial code changes. However, you should be wary of such advice since, while the suggested change might improve the performance, there is no guarantee that it will do so.

Some of the messages are:

- **This loop was not modulo scheduled because it was optimized for space.**

  When a loop is modulo scheduled, it often produces code that has to precede the scheduled loop (the prolog) and code that has to follow the scheduled loop (the epilog). This almost always increases the size of the code. That is why, if you specifies an optimization that minimizes the space requirements, the compiler doesn't attempt modulo scheduling of a loop.

- **This loop was not modulo scheduled because it contains calls or volatile operations**

  Due to the restrictions imposed by calls and volatile memory accesses, the compiler doesn't try to modulo schedule loops containing such instructions.

- **This loop was not modulo scheduled because it contains too many instructions**

  The compiler doesn't try to modulo schedule loops that contain many instructions because the potential for gain is not worth the increased compilation time.

- **This loop was not modulo scheduled because it contains jump instructions**

  Only single block loops are modulo scheduled. You can attempt to restructure your code and use single block loops.

- **Consider using pragma loop_count to specify the trip count or trip modulo**

  This information may help vectorization.

- **Consider using pragma loop_count to specify the trip count or trip modulo, in order to prevent peeling**

  When a loop is vectorized, but the trip count is not known, some iterations are peeled from the loop and executed conditionally (based on the run time value of the trip count). This can be avoided if the trip count is known to be divisible by the number of iterations executed in parallel as a result of vectorization.

- ***operation* of this size is implemented as a library call**

  This message is issued when a source code *operation* results in a library call, due to lack of hardware support for performing that operation on operands of that size.

- ***operation* is implemented as a library call**

  This message is issued when a source code *operation* results in a library call, due to lack of direct hardware support. For instance, an integer division results in a library call.

- **Use of volatile in loops precludes optimizations**

  In general, volatile variables hinder optimizations. They cannot be promoted to registers, because each access to a volatile variable requires accessing the corresponding memory location. The negative effect on performance is amplified if volatile variables are used inside loops. However, there are legitimate cases when you have to use a volatile variable exactly because of this special treatment by the optimizer, for instance when a loop polls if a certain asynchronous condition occurred. This message does not discourage the use of volatile, it just stresses the implications of such a decision.

- **Jumps out of this loop prevent efficient hardware loop generation**

  Due to the presence of jumps out of a loop, the compiler either cannot generate a hardware loop, or was forced to generate one that has a conditional exit.

- **Consider using a 4-byte integral type for the variable *name*, for more efficient hardware loop generation**

  Using short-typed variables as loop control variables limits optimization because the short variables may wrap. For instance, in:

  ```
  unsigned short i;
  for (i = 0; i < c; i++)
      ....
  ```

  if `c` > 65536, then the loop will run forever because `i` wraps from 65535 back to 0. In this case, the compiler must add a wrapper. The compiler recommends using an int variable instead (int or unsigned int) unless the smaller size is critical to your program's behavior.

3 C/C++ RUN-TIME LIBRARY

The C and C++ run-time libraries are collections of functions, macros, and class templates that you can call from your source programs. The libraries provide a broad range of services, including those that are basic to the languages, such as memory allocation, character and string conversions, and math calculations. Using the library simplifies your software development by providing code for a variety of common needs.

This chapter contains

- "C and C++ Run-Time Libraries Guide" on page 3-3
  It provides introductory information about the ANSI/ISO standard C and C++ libraries. It also provides information about the ANSI standard header files and built-in functions that are included with this release of the `ccts` compiler.
- "Documented Library Functions" on page 3-65
  It lists the functions (defined by the C standard header files) that are described in "Run-Time Library Reference" on page 3-69.
- "Run-Time Library Reference" on page 3-69"
  It provides reference information about the C run-time library functions included with this release of the `ccts` compiler.

The `ccts` compiler provides a broad collection of library functions, including those required by the ANSI standard and additional functions supplied by Analog Devices, Inc. that are of value in signal processing. In addition to the standard C library, this release of the compiler software

includes the Abridged C++ library, a conforming subset of the standard C++ library. The Abridged C++ library includes the embedded C++ and embedded standard template libraries.

This chapter describes the standard C/C++ library functions that are supported in the current release of the run-time library as well as a number of signal processing, matrix, and statistical functions that assist project code development.

For more information on the algorithms on which many of the C library's math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Functions*, Englewood Cliffs, New Jersey, Prentice Hall, 1980 (ASIN: 0138220646). For more information on the C++ library portion of the ANSI/ISO Standard for C++, see Plauger, P. J. (Preface), *The Draft Standard C++ Library*, Englewood Cliffs, New Jersey: Prentice Hall, 1994 (ISBN: 0131170031).

The Abridged C++ library software documentation is located on the VisualDSP++ installation CD in the `Docs/Reference` folder. Viewing or printing these files requires a browser, such as Internet Explorer 4.0 (or higher). You can copy these files from the installation CD onto another disk.

C and C++ Run-Time Libraries Guide

The C and C++ run-time libraries contain routines that you can call from your source program. This section describes how to use the libraries and provides information on the following topics:

- "Calling Library Functions" on page 3-4
- "Using Compiler's Built-In C Library Functions" on page 3-4
- "Linking Library Functions" on page 3-5
- "Working With Library Source Code" on page 3-8
- "Working With Library Header Files" on page 3-9
- "DSP Header Files" on page 3-22
- "Calling Library Functions from an ISR" on page 3-31
- "Abridged C++ Library Support" on page 3-32
- "Measuring Cycle Counts" on page 3-39
- "File I/O Support" on page 3-49

The C run-time libraries' functions, available in this `ccts` compiler release are listed in "Documented Library Functions" on page 3-65. For information on the Abridged C++ library's contents, see "Abridged C++ Library Support" on page 3-32 and on-line Help.

Calling Library Functions

To use a C/C++ library function, call the function by name and give the appropriate arguments. The name and arguments for each function appear on the function's reference page. The reference pages appear in the "Run-Time Library Reference" on page 3-69 and in the **C++ Run-Time Library** topic of the on-line Help.

Like other functions you use, library functions should be declared. Declarations are supplied in header files. For more information about the header files, see "Working With Library Header Files" on page 3-9.

Function names are C/C++ function names. If you call a C or C++ run-time library function from an assembly program, you must use the assembly version of the function name (prefix an underscore on the name).

For more information on the naming conventions, see "C/C++ and Assembly Language Interface" on page 1-280.

You can use the archiver, `elfar`, described in the *VisualDSP++ 4.5 Linker and Utilities Manual*, to build library archive files of your own functions.

Using Compiler's Built-In C Library Functions

The C/C++ compiler's built-in functions are a set of functions that the compiler immediately recognizes and replaces with inline assembly code instead of a function call. Typically, inline assembly code is faster than an library routine, and it does not incur the calling overhead.

To use built-in functions, your source must include the required standard include file. For the `abs` functions, this would require `stdlib.h` to be included. Built-in functions are defined for some ANSI C `math.h`, `string.h`, and `stdlib.h` functions. There are also built-in functions to support various Analog Devices extensions to the ANSI standard defined

in the include file `builtins.h`. Not all built-in functions have a library alternate definition. Therefore, the failure to include the required header files may result in your program build failing to link.

If you want to use the C run-time library functions of the same name, compile with the `-no-builtin` compiler switch (see on page 1-45).

Linking Library Functions

The C/C++ run-time library is organized as several libraries which are catalogued in Table 3-1. The libraries and startup files are installed within the subdirectory `. . .\TS\lib` of your VisualDSP++ installation.

Table 3-1. C and C++ Library Files

TS\lib Directory	Description
`libc_TS101.dlb` `libc_TS201.dlb`	C run-time library
`libcpp_TS101.dlb` `libcpp_TS201.dlb`	C++ run-time library
`libcpp_TS101_x.dlb` `libcpp_TS201_x.dlb`	C++ run-time library with exception handling
`libcpprt_TS101.dlb` `libcpprt_TS201.dlb`	C++ run-time support library
`libcpprt_TS101_x.dlb` `libcpprt_TS201_x.dlb`	C++ run-time support library with exception handling
`libdsp_TS101.dlb` `libdsp_TS201.dlb`	DSP run-time library
`libio_TS101.dlb` `libio_TS201.dlb`	I/O run-time library
`libsim.dlb` `libsim_TS201.dlb`	Simulator library support
`libx_TS101.dlb` `libx_TS201.dlb`	C++ exception handling support library

Table 3-1. C and C++ Library Files (Cont'd)

TS\lib Directory	Description
ts_hdr_TS101.doj ts_hdr_TS201.doj	C startup file – calls setup routine and main()
ts_hdr_cpp_TS101.doj ts_hdr_cpp_TS201.doj	C++ start-up file -- calls setup routine and main()
ts_exit_TS101.doj ts_exit_TS201.doj	C exit routine
ts_exit_cpp_TS101.doj ts_exit_cpp_TS201.doj	C++ exit routine
meminit_TS101.doj meminit_TS201.doj	Memory initialization support

In general, several versions of the libraries and start-up files are supplied in binary form. For instance, the DSP run-time library is built twice; once for the ADSP-TS101 processor and once for the ADSP-TS20x processors.

Binary files that are built for the ADSP-TS101 processor have a _TS101 suffix and binary files built for the ADSP-TS20x processors have a _TS201 suffix.

Other variants of the library and startup files are supplied that are compatible with applications that have been compiled with the compiler switch -char-size-8 (on page 1-28); these binary files have a _ba suffix. File names that do not have a _ba suffix have been built for applications that use the conventional word-addressing mode.

Binary files that have a _mt suffix have been built for multi-threaded environments; file names that do not have a _mt suffix have been built for a single-threaded environment.

The libraries located in . . .\TS\lib are built without any workarounds enabled. There are subdirectories within the . . .\TS\lib directory named <target>_rev_<revision> that contain libraries built for <target>

(ts101, ts20x) and for the specific revision, with the appropriate workarounds for the specific silicon revision. An example would be the directory "ts101_rev_0.0".

One single library directory may support more than one specific silicon revision, for example the "ts101_rev_0.0" libraries are valid for revisions 0.0 and 0.1 of the ADSP-TS101 processor. A set of libraries is included that supports all suitable workarounds valid for any of the revisions of a particular target.

By default, the libraries built with workarounds enabled for the most recently-supported version of the processor are used. There is no special suffix for the libraries built with workarounds enabled.

The -si-revision switch (on page 1-61) can be used to specify a silicon revision—VisualDSP++ will use the appropriately-built library when linking the application.

When you call a run-time library function, the call creates a reference that the linker resolves. One way to direct the linker to the library's location is to use the default Linker Description File (ADSP-TS<your_target>.ldf).

If you are not using the default .LDF file, then either add the appropriate library/libraries to the .LDF file used for your project or use the compiler's -l switch to specify the library to be added to the link line. For example, the switches -lc_TS201 -ldsp_TS201 add the libraries libc_TS201.dlb and libdsp_TS201.dlb to the list of libraries to be searched by the linker. For more information on the .LDF file, see the *VisualDSP++ 4.5 Linker and Utilities Manual*.

If all the objects supplied to the driver have been built as C, but are referencing a C++ object which is in a library, the standard C++ libraries are not searched and the linker may issue an error concerning unresolved symbol(s). This can be avoided by using the compiler flags-link switch (see on page 1-35), which ensures that the C++ libraries are linked from the default .LDF files. For example, -flags-link -MD__cplusplus=1.

Note that this problem only occurs if the C++ object is in a library. If it is in an object file, the compiler recognizes it as a C++ object and links with the C++ libraries.

Working With Library Source Code

The source code for the functions in the C and DSP run-time libraries is provided with your VisualDSP++ software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept, named `...\TS\lib\src`. The directory contains the source for the C run-time library, for the DSP run-time library, and for the I/O run-time library, as well as the source for the main program start-up functions. If you do not intend to modify any of the run-time library functions and are not interested in using the source code as a reference, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize any particular function for your own needs. To modify these files, you need proficiency in the TigerSHARC assembly language and an understanding of the run-time environment, as explained in "C/C++ Run-Time Model and Environment" on page 1-252. Before you make any modifications to the source code, copy the source code to a file with a different filename and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct.

Analog Devices supports the run-time library functions only as provided.

Working With Library Header Files

When you use a library function in your program, you should also include the function's header file with the `#include` preprocessor command. The header file for each function is identified in the **Synopsis** section of the function's reference page. Header files contain function prototypes. The compiler uses these prototypes to check that each function is called with the correct arguments.

The header files define the non-suffixed names (for example, `sin`) as the 32-bit versions (for example, `sinf`) if the compiler is treating doubles as 32 bits or as the 64-bit version (for example, `sind`). This lets you use the non-suffixed names with arguments of type `double`, regardless of whether doubles are 32 or 64 bits.

The routines suffixed with `f` always require 32-bit arguments, and the routines suffixed with `d` always require 64-bit arguments, regardless of whether the `double` type is 32 or 64 bits.

Some of these routines set the `errno` global variable, as required by the C standard. To use `errno`, you must include the header file `errno.h`.

By default, the `ccts` compiler makes the `double` type 32 bits long, for high performance. You can select a compiler option to make the `double` type 64 bits long for conformance with the C standard, but this makes computations with double values slower. In either case, the `float` type is 32 bits, and the `long double` type is 64 bits.

A list of the header files that are supplied with this release of the `ccts` compiler appears in Table 3-2. You should use a C standard text to augment the information supplied in this chapter.

The following are the descriptions of the header files contained in the C library. The header files are listed in alphabetical order.

Table 3-2. Standard C Run-Time Library Header Files

Header	Purpose	Standard
assert.h	Diagnostics	ANSI
ctype.h	Character Handling	ANSI
cycle_count.h	Basic Cycle Counting	Analog extension
cycles.h	Cycle Counting with Statistics	Analog extension
device.h	Macros and data structures for alternative device drivers	Analog extension
device_int.h	Enumerations and prototypes for alternative device drivers	Analog extension
errno.h	Error Handling	ANSI
float.h	Floating Point	ANSI
iso646.h	Boolean Operators	ANSI
limits.h	Limits	ANSI
locale.h	Localization	ANSI
math.h	Mathematics	ANSI
setjmp.h	Non-Local Jumps	ANSI
signal.h	Signal Handling	ANSI
stdarg.h	Variable Arguments	ANSI
stddef.h	Standard Definitions	ANSI
stdio.h	Input/Output	ANSI
stdlib.h	Standard Library	ANSI
string.h	String Handling	ANSI
time.h	Date and Time	ANSI

assert.h

The `assert.h` header file defines the `assert` macro, which can be used to insert run-time diagnostics into a source file. The macro normally tests (asserts) that an expression is true. If the expression is false, then the macro

will first print an error message, and will then call the abort function to terminate the application. The message displayed by the assert macro will be of the form:

```
filename : linenumber assertion failed:“ expression“
```

Note that the message includes the following information:

- `filename` - the name of the source file
- `linenumber` - the current line number in the source file
- `expression` - the expression tested

However if the macro `NDEBUG` is defined at the point at which the `assert.h` header file is included in the source file, then the assert macro will be defined as a null macro and no run-time diagnostics will be generated.

ctype.h

The `ctype.h` header file contains functions for character handling, such as `isalpha`, `tolower`, etc.

cycle_count.h

The `cycle_count.h` header file provides an inexpensive method for benchmarking C-written source by defining basic facilities for measuring cycle counts. The facilities provided are based upon two macros, and a data type which are described in more detail in the section “Measuring Cycle Counts” on page 3-39.

cycles.h

The `cycles.h` header file defines a set of five macros and an associated data type that may be used to measure the cycle counts used by a section of C-written source. The macros can record how many times a particular piece of code has been executed and also the minimum, average, and max-

imum number of cycles used. The facilities that are available via this header file are described in the section "Measuring Cycle Counts" on page 3-39.

device.h

The `device.h` header file provides macros and defines data structures that an alternative device driver would require to provide file input and output services for `stdio` library functions. Normally, the `stdio` functions use a default driver to access an underlying device, but alternative device drivers may be registered that may then be used transparently by these functions. This mechanism is described in "Extending I/O Support To New Devices" on page 3-50.

device_int.h

The `device_int.h` header file contains function prototypes and provides enumerations for alterative device drivers. An alternative device driver is normally provided by an application and may be used by the `stdio` library functions to access an underlying device; an alternative device driver may coexist with, or may replace, the default driver that is supported by the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Refer to "Extending I/O Support To New Devices" on page 3-50 for more information.

errno.h

The `errno.h` header file provides access to `errno` and also defines macros for associated error codes. This facility is not, in general, supported by the rest of the library.

float.h

The `float.h` header file defines the properties of the floating-point data types that are implemented by the compiler—that is, `float`, `double`, and `long double`. These properties are defined as macros and include the following for each supported data type:

- the maximum and minimum value (for example, `FLT_MAX` and `FLT_MIN`)
- the maximum and minimum power of ten (for example, `FLT_MAX_10_EXP` and `FLT_MIN_10_EXP`)
- the precision available expressed in terms of decimal digits (for example, `FLT_DIG`)
- a constant that represents the smallest value that may added to 1.0 and still result in a change of value (for example, `FLT_EPSILON`)

Note that the set of macros that define the properties of the `double` data type will have the same values as the corresponding set of macros for the `float` type when `doubles` are defined to be 32 bits wide, and they will have the same value as the macros for the `long double` data type when `doubles` are defined to be 64 bits wide (see "-double-size-{32 | 64}" on page 1-31).

iso646.h

The `iso646.h` header file defines symbolic names for certain C operators; the symbolic names and their associated value are shown in Table 3-3.

Table 3-3. Symbolic Names Defined in iso646.h

Symbolic Name	Equivalent
`and`	&&
`and_eq`	&=
`bitand`	&
`bitor`	|
`compl`	~
`not`	!
`not_eq`	!=
`or`	||
`or_eq`	|=
`xor`	^
`xor_eq`	^=

The symbolic names have the same name as the C++ keywords that are accepted by the compiler when the `-alttok` switch (see on page 1-25) is specified.

limits.h

The `limits.h` header file contains definitions of maximum and minimum values for each C data type other than floating-point.

locale.h

The `locale.h` header file contains definitions for expressing numeric, monetary, time, and other data.

math.h

The `math.h` header file includes trigonometric, power, logarithmic, exponential, and other miscellaneous functions. The library contains the functions specified by the C standard along with implementations for the data types `float` and `long double`.

For every function that is defined to return a `double`, the `math.h` header file also defines corresponding functions that return a `float` and a `long double`. The names of the `float` functions are the same as the equivalent `double` function with `f` appended to its name. Similarly, the names of the `long double` functions are the same as the `double` function with `d` appended to its name. For example, the header file contains the following prototypes for the sine function:

```
float sinf (float x);
double sin (double x);
long double sind (long double x);
```

When the compiler is treating `double` as 32 bits, the header file arranges that all references to the `double` functions are directed to the equivalent `float` function (with the suffix `f`). This allows the un-suffixed function names to be used with arguments of type `double`, regardless of whether doubles are 32 or 64 bits long.

This header file also provides prototypes for a number of additional math functions provided by Analog Devices, such as `favg`, `fmax`, `fclip`, and `copysign`.

Some of the functions in this header file exist as both integer and floating point. The floating-point functions typically have an `f` prefix. Make sure you are using the correct one. The C language provides for implicit type conversion, so the following sequence produces surprising results with no warnings:

```
float x,y;
y = abs(x);
```

The value in `x` is truncated to an integer prior to calculating the absolute value, then reconverted to floating point for the assignment to `y`.

A number of functions (including `fabs`, `favg`, `fmax`, `fmin`, `fclip`, and `copysign`) are implemented via intrinsics (provided the header file has been `#include` 'd) that map to a single machine instruction.

If the header is not included, the library implementation is used instead, at a considerable loss in efficiency.

Individual function descriptions focus on the 32-bit `float` version. When the full range is given for **Domain**, the 64-bit `float` interfaces are not limited to 3.4×10^{38}.

setjmp.h

The `setjmp.h` header file contains `setjmp` and `longjmp` for non-local jumps.

signal.h

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several extensions, such as `interrupt()`.

The signal handling functions process conditions (hardware signals) that can occur during program execution. They determine the way that your C program responds to these signals. The functions are designed to process such signals as external interrupts and timer interrupts.

stdarg.h

The `stdarg.h` header file contains definitions needed for functions that accept a variable number of arguments. Programs that call such functions must include a prototype for the functions referenced.

stddef.h

The `stddef.h` header file contains a few common definitions useful for portable programs, such as `size_t`.

stdio.h

The `stdio.h` header file defines a set of functions, macros, and data types for performing input and output. Applications that use the facilities of this header file should link with the I/O library in the same way as linking with the C run-time library (see "Linking Library Functions" on page 3-5). The library is thread-safe but it is not interrupt-safe and should not therefore be called either directly or indirectly from an interrupt service routine.

The compiler uses definitions within the header file to select an appropriate set of functions that correspond to the currently selected addressing mode (either word-addressing or byte-addressing) and size of type `double` (either 32 bits or 64 bits). Any source file that uses the facilities of `stdio.h` must therefore include the header file. Failure to include the header file results in a linker failure as the compiler must see a correct function prototype in order to generate the correct calling sequence.

The implementation of the `stdio.h` routines is based on a simple interface with a device driver that provides a set of low-level primitives for `open`, `close`, `read`, `write`, and `seek` operations. By default, these operations are provided by the VisualDSP++ simulator and EZ-KIT Lite systems and this mechanism is outlined in "Default Device Driver Interface" on page 3-58.

Alternative device drivers may be registered that can then be used transparently through the `stdio.h` functions. See "Extending I/O Support To New Devices" on page 3-50 for a description of the feature.

The following restrictions apply to this software release:

- the functions `tmpfile` and `tmpnam` are not available
- the functions `rename` and `remove` are supported only under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite systems, and they only operate on the host file system
- positioning within a file that has been opened as a text stream is supported only if the lines within the file are terminated by the character sequence `\r\n`
- Support for formatted reading and writing of data of type `long double` is supported only if an application is built with the `-double-size-64` switch

At program termination, the host environment closes down any physical connection between the application and an opened file. However, the I/O library does not implicitly close any opened streams to avoid unnecessary overheads (particularly with respect to memory occupancy). Thus, unless explicit action is taken by an application, any unflushed output may be lost. Any output generated by `printf` is always flushed. However, output generated by other library functions, such as `putchar`, `fwrite`, `fprintf`, is not automatically flushed. Applications should therefore arrange to close down any streams that they open. Note that the function reference `fflush(NULL);` flushes the buffers of all opened streams.

Each opened stream is allocated a buffer which either contains data from an input file or output from a program. For text streams, this data is held in the form of 8-bit characters that are packed into 32-bit memory locations. Due to internal mechanisms used to unpack and pack this data, the buffer must not reside at a memory location that is greater than the address `0x3fffffff`. Since the `stdio` library allocates buffers from the heap, this restriction implies that the heap should not be placed at address `0x40000000` or above.

The restriction may be avoided by using the `setvbuf` function to allocate the buffer from alternative memory, as in the following example (that assumes that the buffer resides at a memory location that is less than `0x40000000`).

```
#include <stdio.h>

char buffer[BUFSIZ];
setvbuf(stdout,buffer,_IOLBF,BUFSIZ);
printf("Hello World\n");
```

A faster set of functions is available for applications that print only to standard output. These functions are linked into an application if you compile with the switch `-flags-link -MD__USING_LIBSIM=1`. This switch forces the linker to link against the library `libsim.dlb`. This library contains a limited set of `stdio.h` facilities that are executed on the host of the VisualDSP++ debugger rather than inside the debugger's target. This type of execution leads to smaller applications and faster output.

The following functions are supported by the `libsim.dlb` library for this compiler release only.

```
printf
sprintf
fprintf
```

All three of these functions from `libsim.dlb` use the `L` modifier, as in `%LF`, to specify a conversion for a `long double` (64-bit floating-point) argument. They also use the "`ll`" modifier, as in `%lld`, to specify a conversion for a `long long` (64-bit integer) argument. The `fprintf` function in this library currently ignores its `FILE*` stream argument and always prints to standard output.

The libsim.dlb library is not available in byte-addressing mode.

stdlib.h

The `stdlib.h` header file contains general utilities specified by the C standard. These include some integer math functions, such as `abs`, `div`, and `rand`; general string-to-numeric conversions; memory allocation functions, such as `malloc` and `free`; and termination functions, such as `exit`. This header file also contains prototypes for miscellaneous functions, such as `bsearch` and `qsort`.

This header also provides prototypes for a number of additional integer math functions provided by Analog Devices, such as `avg`, `max`, `clip`; also `count_ones` and `addbitrev` .

Some of the functions included in this file exist as both integer and floating point. The floating-point functions typically have an `f` prefix. Make sure you are using the correct type.

A number of functions (including `abs`, `avg`, `max`, `min`, `clip`, `count_ones`, and `addbitrev`) are implemented via intrinsics (provided the header file has been `#include`'d) which map to single machine instructions.

If the header is not included, the library implementation is used instead, at a considerable loss in efficiency.

string.h

The `string.h` header file contains string handling functions, including `strcpy` and `memcpy`.

time.h

The `time.h` header file provides functions, data types, and a macro for expressing and manipulating date and time information. The header file defines two fundamental data types, one of which is `clock_t` and is associated with the number of implementation-dependent processor “ticks” used since an arbitrary starting point; and the other which is `time_t`.

The `time_t` data type is used for values that represent the number of seconds that have elapsed since a known epoch; values of this form are known as a *calendar time*. In this implementation, the epoch starts on 1st January, 1970, and calendar times before this date are represented as negative values.

A calendar time may also be represented in a more versatile way as a broken-down time. A broken-down time is a structured variable of the following form:

```
struct tm { int tm_sec; /* seconds after the minute [0,61] */
  int tm_min;           /* minutes after the hour [0,59]   */
  int tm_hour;          /* hours after midnight [0,23]     */
  int tm_mday;          /* day of the month [1,31]         */
  int tm_mon;           /* months since January [0,11]     */
  int tm_year;          /* years since 1900                */
  int tm_wday;          /* days since Sunday [0, 6]        */
  int tm_yday;          /* days since January 1st [0,365]  */
  int tm_isdst;         /* Daylight Saving flag            */
};
```

This implementation does not support either the Daylight Saving flag in the structure struct tm; nor does it support the concept of time zones. All calendar times are therefore assumed to relate to Greenwich Mean Time (Coordinated Universal Time or UTC).

The header file sets the `CLOCKS_PER_SEC` macro to the number of processor cycles per second and this macro can therefore be used to convert data of type `clock_t` into seconds, normally by using floating-point arithmetic to divide it into the result returned by the `clock` function.

In general, the processor speed is a property of a particular chip and it is therefore recommended that the value to which this macro is set is verified independently before it is used by an application.

In this version of the C/C++ compiler, the `CLOCKS_PER_SEC` macro is set by one of the following (in descending order of precedence):

- Via the `-DCLOCKS_PER_SEC=<definition>` compile-time switch
- Via the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- From the header file `cycles.h`

DSP Header Files

The DSP header files contains prototypes for all the DSP library functions. When the appropriate `#include` preprocessor command is included in your source, the compiler uses the prototypes to check that each function is called with the correct arguments.

The DSP header files included in this release of the `ccts` compiler are:

- "complex.h – Basic Complex Arithmetic Functions" on page 3-23
- "filter.h – DSP Filters and Transformations" on page 3-24
- "libsim.h – Simulator Services" on page 3-25
- "matrix.h – Matrix Functions" on page 3-26
- "stats.h – Statistical Functions" on page 3-27
- "vector.h – Vector Functions" on page 3-28
- "window.h – Window Generators" on page 3-29

complex.h – Basic Complex Arithmetic Functions

The `complex.h` header file contains type definitions and basic arithmetic operations for variables of type `complex_float`, `complex_double`, and `complex_long_double`. The complex functions are listed with their 32

bits float interface (for example, `float` in C) only. The old style K&R form of representing the parameters is used in this section, for convenience purposes only.

Refer to Table 3-7 for a list of complex functions which are described in detail in "Run-Time Library Reference" on page 3-69.

The following structures are used to represent complex numbers in rectangular coordinates:

```
typedef struct
{
   float re;
   float im;
} complex_float;

typedef struct
{
   double re;
   double im;
} complex_double;

typedef struct
{
   long double re;
   long double im;
} complex_long_double;
```

filter.h – DSP Filters and Transformations

The `filter.h` header file contains filters used in signal processing. It also includes the A-law and µ-law companders that are used by voice-band compression and expansion applications. This header file also contains functions that perform key signal processing transformations, including Fast Fourier Transforms (FFT) and convolution.

Various forms of the FFT function are provided by the library corresponding to radix-2, radix-4, and two-dimensional FFTs. The number of points is provided as an argument, and the library uses radix-2 or radix-4 implementations as appropriate.

The library also provides two optimized complex and real FFT functions that have been implemented using a fast radix-2 algorithm; however these functions, `cfftf` and `rfftf`, have certain requirements that may not be appropriate for some applications.

The functions defined in the `filter.h` header file are listed in Table 3-8 and are described in detail in "Run-Time Library Reference" on page 3-69.

Library functions are provided to initialize a twiddle table for an FFT function. A twiddle table is normally calculated once during program initialization and can be used to accommodate several FFTs of different sizes by allocating the table at maximum size, and then using the stride argument of the FFT function to specify the step size through the table. If the stride argument is set to 1, the FFT function uses all the table; if the FFT uses only half the number of points of the largest, the stride should be 2.

The functions described by this header make certain assumptions about their arguments, in order to achieve high efficiency:

- The FFT routines require that the input, output, and temporary arrays be quad-word aligned, and the twiddle table be long-word aligned.
- The filter routines require that the input array (and coefficients) for FIR be quad-word aligned.
- The A-law and μ-law companders require that the input and output arrays be quad-word aligned.

Failure to observe these constraints results in incorrect operation.

libsim.h – Simulator Services

The `libsim.h` header file defines services that are provided by the VisualDSP++ debugging environment. The header file contains the function `__emuclk`, which returns the current simulator cycle count. The header file also contains several utility print routines that may be useful for assembly language development since they are much easier to call than `printf`. Table 3-4 lists the utility print routines and provides a brief description of each.

The list of routines contains names suitable for calling from a C program. If you call them from an assembly program, add an additional leading underscore to the name.

In addition, these functions conform to the C calling conventions, as described in "C/C++ Run-Time Model and Environment" on page 1-252.

Table 3-4. libsim Print Routines

Function	Description
`__print_int`	prints 32-bit int
`__print_uint`	prints unsigned 32-bit int
`__print_float`	prints 32-bit float
`__print_double`	prints 64-bit long double float
`__print_longlong`	prints 64-bit signed int
`__print_ulonglong`	prints 64-bit unsigned int

The libsim.h functionality is not available in byte-addressing mode.

matrix.h – Matrix Functions

The `matrix.h` header file contains matrix functions for operating on real and complex matrices, both matrix-scalar and matrix-matrix operations. See "complex.h – Basic Complex Arithmetic Functions" on page 3-23 for definitions of the complex types.

Refer to Table 3-10 for a list of matrix functions which are described in detail in "Run-Time Library Reference" on page 3-69.

The matrix functions are listed with their 32-bit float interface (for example, `float` in C) only. The old style K&R form of representing the parameters is used in this section for convenience purposes only. The matrix functions are each provided in three forms—for the three floating-point types. For brevity, only the `float` form is listed in the detailed descriptions.

For example, under the `cvecsadd` function, the following list is provided in the **Synopsis**:

```
cvecsaddf(float a[], float b, float c[], int n)
```

But, the library also provides:

```
cvecsadd(double a[], double b, double c[], int n)

cvecsaddd(long double a[], long double b, long double c[], int n)
```

The functions described by this header make certain assumptions about their arguments, in order to achieve high efficiency:

- Input array arguments are constant; for example, their contents do not change during the course of the routine. In particular, this means the input arguments do not overlap with any output argument.

- Input array arguments are quad-word aligned, and output array arguments are at least double-word aligned. The compiler provides this alignment for all top-level arrays; you must not pass an argument which points at an arbitrary, non-aligned location in an array.

Failure to observe these constraints results in incorrect operation.

stats.h – Statistical Functions

The `stats.h` header file contains statistical functions, such as `autocoh` and `crosscoh`. The statistics routines make the following assumptions about their arguments:

- Input array arguments are constant; for example, their contents do not change during the course of the routine. In particular, this means the input arguments do not overlap with any output arguments.
- Input array arguments are quad-word aligned, and output array arguments are at least double-word aligned. The compiler provides this alignment for all top-level arrays; you must not pass an argument that points at an arbitrary, non-aligned location in an array.

Failure to observe these constraints results in incorrect operation.

Refer to Table 3-12 for a list of statistical functions that are described in detail in "Run-Time Library Reference" on page 3-69.

vector.h – Vector Functions

The `vector.h` header file contains functions for operating on real and complex vectors, both vector-scalar and vector-vector operations. See "complex.h – Basic Complex Arithmetic Functions" on page 3-23 for the definitions of the complex types.

Refer to Table 3-16 for a list of vector functions which are described in detail in "Run-Time Library Reference" on page 3-69.

The vector functions are listed with their 32-bit float interface (for example, `float` in C) only. The old style K&R form of representing the parameters is used in this section for convenience purposes only.

The vector functions are each provided in three forms—for the three floating-point types. For brevity, only the `float` form is listed in the detailed descriptions. For example, under the `cvecsadd` function, the following list is provided in the **Synopsis**.

```
cvecsaddf(float a[], float b, float c[], int n)
```

but the library also provides

```
cvecsadd(double a[], double b, double c[], int n)
cvecsaddd(long double a[], long double b, long double c[], int n)
```

The functions described by this header make certain assumptions about their arguments, in order to achieve high efficiency:

- Input array arguments are constant; that is, their contents do not change during the course of the routine. In particular, this means the input arguments do not overlap with any output argument.
- Input array arguments are quad-word aligned, and output array arguments are at least double-word aligned. The compiler provides this alignment for all top-level arrays; do not pass an argument which points at an arbitrary, non-aligned location in an array.

Failure to observe these constraints results in incorrect operation.

window.h – Window Generators

The `window.h` header file contains various functions to generate windows based on various methodologies. The functions, defined in the `window.h` header file, are listed in Table 3-5 on page 3-30.

For all window functions, a stride parameter `a` can be used to space the window values. The window length parameter `n` equates to the number of elements in the window. Therefore, for a stride `a` of 2 and a length `n` of 10, an array of length 20 is required, where every second entry is untouched.

Table 3-5. Window Generator Functions

Description	Prototype
generate bartlett window	`void gen_bartlett (float w[], int a, int N)`
generate blackman window	`void gen_blackman (float w[], int a, int N)`
generate gaussian window	`void gen_gaussian (float w[], float alpha, int a, int N)`
generate hamming window	`void gen_hamming (float w[], int a, int N)`
generate hanning window	`void gen_hanning (float w[], int a, int N)`
generate harris window	`void gen_harris (float w[], int a, int N)`
generate kaiser window	`void gen_kaiser (float w[], float beta, int a, int N)`
generate rectangular window	`void gen_rectangular (float w[], int a, int N)`
generate triangle window	`void gen_triangle (float w[], int a, int N)`
generate von hann window	`void gen_vonhann (float w[], int a, int N)`

Calling Library Functions from an ISR

Not all C run-time library functions are interrupt-safe (and can therefore be called from an Interrupt Service Routine). For a run-time function to be classified as *interrupt-safe*, it must:

- not update any global data, such as `errno`, and
- not write to (or maintain) any private static data

It is recommended therefore that none of the functions defined in the header file `math.h`, nor the string conversion functions defined in `stdlib.h`, be called from an ISR as these functions are commonly defined to update the global variable `errno`. Similarly, the functions defined in the `stdio.h` header file maintain static tables for currently opened streams and should not be called from an ISR. Additionally, the memory allocation routines malloc, calloc, realloc, free, and the C++ operators new and delete read and update global tables and are not interrupt-safe.

Several other library functions are not interrupt-safe because they make use of private static data. These functions are:

```
asctime
gmtime
localtime
rand
srand
strtok
```

While not all C run-time library functions are interrupt-safe, versions of the functions are available that are *thread-safe* and may be used in a multi-threaded environment. These library functions can be found in the run-time libraries that have the suffix `_mt` in their filename.

Abridged C++ Library Support

When in C++ mode, the `ccts` compiler can call a large number of functions from the Abridged Library, a conforming subset of the C++ library.

The Abridged C++ library has two major components: embedded C++ library (EC++) and embedded standard template library (ESTL). The embedded C++ library is a conforming implementation of the embedded C++ library as specified by the Embedded C++ Technical Committee. You can view the Abridged Library Reference by locating the file `docs/cpl_lib/index.html` underneath your `VisualDSP++` installation and opening it in a web browser.

This section lists and briefly describes the following components of the Abridged C++ library:

- "Embedded C++ Library Header Files" on page 3-32
- "C++ Header Files for C Library Facilities" on page 3-35
- "Embedded Standard Template Library Header Files" on page 3-36
- "Using the Thread-Safe C/C++ Run-Time Libraries with VDK" on page 3-39

For more information on the Abridged Library, see online Help.

Embedded C++ Library Header Files

The following sections provide a brief description of the header files in the embedded C++ library.

complex

The `complex` header defines the template class `complex` and a set of associated arithmetic operators. Predefined types include `complex_float` and `complex_long_double`.

This implementation does not support the full set of complex operations as specified by the C++ standard. In particular, it does not support either the transcendental functions or the I/O operators << and >>.

The `complex` header and the C library header file `complex.h` refer to two different and incompatible implementations of the `complex` data type.

exception

The `exception` header defines the `exception` and `bad_exception` classes and several functions for exception handling.

fract

The `fract` header defines the `fract` data type, which supports fractional arithmetic, assignment, and type-conversion operations. The header file is fully described under "C++ Fractional Type Support" on page 1-234. An example that demonstrates its use appears in "C++ Programming Examples" on page 1-286.

fstream

The `fstream` header defines the `filebuf`, `ifstream`, and `ofstream` classes for external file manipulations.

iomanip

The `iomanip` header declares several `iostream` manipulators. Each manipulator accepts a single argument.

ios

The `ios` header defines several classes and functions for basic `iostream` manipulations.

Most of the `iostream` header files include `ios`.

iosfwd

The `iosfwd` header declares forward references to various `iostream` template classes defined in other standard headers.

iostream

The `iostream` header declares most of the `iostream` objects used for the standard stream manipulations.

istream

The `istream` header defines the `istream` class for `iostream` extractions.

Most of the iostream header files include `istream`.

new

The `new` header declares several classes and functions for memory allocations and deallocations.

ostream

The `ostream` header defines the `ostream` class for `iostream` insertions.

sstream

The `sstream` header defines the `stringbuf`, `istringstream`, and `ostringstream` classes for various `string` object manipulations.

stdexcept

The `stdexcept` header defines a variety of classes for `exception` reporting.

streambuf

The `streambuf` header defines the `streambuf` classes for basic operations of the `iostream` classes.

Most of the `iostream` header files include `streambuf`.

string

The `string` header defines the `string` template and various supporting classes and functions for `string` manipulations.

Objects of the `string` type should not be confused with the null-terminated C strings.

strstream

The `strstream` header defines the `strstreambuf`, `istrstream`, and `ostream` classes for `iostream` manipulations on allocated, extended, and freed character sequences.

C++ Header Files for C Library Facilities

For each C standard library header file there is a corresponding standard C++ header. If the name of a C standard library header file is `foo.h`, then the name of the equivalent C++ header file is `cfoo`. For example, the C++ header file `cstdio` provides the same facilities as the C header file `stdio.h`.

Table 3-6 lists the C++ header files that provide access to the C library facilities.

The C standard headers files may be used to define names in the C++ global namespace, while the equivalent C++ header files define names in the standard namespace.

Table 3-6. C++ Header Files for C Library Facilities

Header	Description
cassert	Enforces assertions during function executions
cctype	Classifies characters
cerrno	Error codes reported by library functions
cfloat	Floating-point type properties
climits	Integer type properties
clocale	Adapts to different cultural conventions
cmath	Provides common mathematical operations
csetjmp	Executes non-local goto statements
csignal	Controls various exceptional conditions
cstdarg	Accesses a variable number of arguments
cstddef	Defines several useful data types and macros
cstdio	Performs input and output
cstdlib	Performs a variety of operations
cstring	Manipulates several kinds of strings

Embedded Standard Template Library Header Files

Templates and the associated header files are not part of the embedded C++ standard, but they are supported by the `ccts` compiler in C++ mode.

The embedded standard template library headers are listed below.

algorithm

The `algorithm` header defines numerous common operations on sequences.

deque

The `deque` header defines a deque template container.

functional

The `functional` header defines numerous function objects.

hash_map

The `hash_map` header defines two hashed map template containers.

hash_set

The `hash_set` header defines two hashed set template containers.

iterator

The `iterator` header defines common iterators and operations on iterators.

list

The `list` header defines a list template container.

map

The `map` header defines two map template containers.

memory

The `memory` header defines facilities for managing memory.

numeric

The `numeric` header defines several numeric operations on sequences.

queue

The `queue` header defines two queue template container adapters.

set

The `set` header defines two set template containers.

stack

The `stack` header defines a stack template container adapter.

utility

The `utility` header defines an assortment of utility templates.

vector

The `vector` header defines a vector template container.

The embedded C++ library also includes several header files for compatibility with traditional C++ libraries, such as:

fstream.h

The `fstream.h` header file defines several `iostreams` template classes that manipulate external files.

iomanip.h

The `iomanip.h` header file declares several `iostreams` manipulators that take a single argument.

iostream.h

The `iostream.h` header file declares the `iostreams` objects that manipulate the standard streams.

new.h

The `new.h` header file declares several functions that allocate and free storage.

Using the Thread-Safe C/C++ Run-Time Libraries with VDK

When developing for VDK, the thread-safe variants of the run-time libraries are linked with user applications. These libraries may add an overhead to the VDK resources required by some applications.

The run-time libraries make use of VDK synchronicity functions to ensure thread safety.

Measuring Cycle Counts

The common basis for benchmarking some arbitrary C-written source is to measure the number of processor cycles that the code uses. Once this figure is known, it can be used to calculate the actual time taken by multiplying the number of processor cycles by the clock rate of the processor. The run-time library provides three alternative methods for measuring processor counts. Each of these methods is described in:

- "Basic Cycle Counting Facility" on page 3-40
- "Cycle Counting Facility with Statistics" on page 3-41
- "Using time.h to Measure Cycle Counts" on page 3-44
- "Determining the Processor Clock Rate" on page 3-46
- "Considerations When Measuring Cycle Counts" on page 3-47

Basic Cycle Counting Facility

The fundamental approach to measuring the performance of a section of code is to record the current value of the cycle count register before executing the section of code, and then reading the register again after the code has been executed. This process is represented by two macros that are defined in the `cycle_count.h` header file; the macros are:

```
START_CYCLE_COUNT(S)
```

```
STOP_CYCLE_COUNT(T,S)
```

The parameter `S` is set by the macro `START_CYCLE_COUNT` to the current value of the cycle count register; this value should then be passed to the macro `STOP_CYCLE_COUNT`, which will calculate the difference between the parameter and current value of the cycle count register. Reading the cycle count register incurs an overhead of a small number of cycles and the macro ensures that the difference returned (in the parameter `T`) will be adjusted to allow for this additional cost. The parameters `S` and `T` may be separate variables but more efficient code will be used if they represent the same variable; they should be declared as a `cycle_t` data type which the `cycle_count.h` header file defines as:

```
typedef volatile unsigned long long cycle_t;
```

The header file also defines the macro:

```
PRINT_CYCLES(STRING,T)
```

which is provided mainly as an example of how to print a value of type `cycle_t`; the macro outputs the text `STRING` on `stdout` followed by the number of cycles `T`.

The instrumentation represented by the macros defined in this section is only activated if the program is compiled with the `DO_CYCLE_COUNTS` macro defined. If this macro is not specified, then the macros are replaced by empty statements and have no effect on the program.

The following example demonstrates how the basic cycle counting facility may be used to monitor the performance of a section of code:

```
#include <cycle_count.h>
#include <stdio.h>

extern int
main(void)
{
   cycle_t cycle_count;
```

```
    START_CYCLE_COUNT(cycle_count)
    Some_Function_Or_Code_To_Measure();
    STOP_CYCLE_COUNT(cycle_count,cycle_count)

    PRINT_CYCLES("Number of cycles: ",cycle_count)
}
```

The run-time libraries provide alternative facilities for measuring the performance of C source (see "Cycle Counting Facility with Statistics" on page 3-41 and "Using time.h to Measure Cycle Counts" on page 3-44); the relative benefits of this facility are outlined in "Considerations When Measuring Cycle Counts" on page 3-47.

The basic cycle counting facility is based upon macros; it may therefore be customized for a particular application if required, without the need for rebuilding the run-time libraries.

Cycle Counting Facility with Statistics

The `cycles.h` header file defines a set of macros for measuring the performance of compiled C source. As well as providing the basic facility for reading the cycle count registers of the TigerSHARC architecture, the macros also have the capability of accumulating statistics that are suited to recording the performance of a section of code that is executed repeatedly.

If the macro `DO_CYCLE_COUNTS` is specified at compile-time, then the `cycles.h` header file defines the following macros:

- `CYCLES_INIT(S)`
  a macro that initializes the system timing mechanism and clears the parameter `S`; an application must contain one reference to this macro.
- `CYCLES_START(S)`
  a macro that extracts the current value of the cycle count register and saves it in the parameter `S`.

- `CYCLES_STOP(S)`
  a macro that extracts the current value of the cycle count register and accumulates statistics in the parameter `S`, based on the previous reference to the `CYCLES_START` macro.

- `CYCLES_PRINT(S)`
  a macro which prints a summary of the accumulated statistics recorded in the parameter `S`.

- `CYCLES_RESET(S)`
  a macro which re-zeros the accumulated statistics that are recorded in the parameter `S`.

The parameter `S` that is passed to the macros must be declared to be of the type `cycle_stats_t`; this is a structured data type that is defined in the `cycles.h` header file. The data type has the capability of recording the number of times that an instrumented part of the source has been executed, as well as the minimum, maximum, and average number of cycles that have been used. If an instrumented piece of code has been executed for example, 4 times, the `CYCLES_PRINT` macro would generate output on the standard stream `stdout` in the form:

```
AVG   : 95
MIN   : 92
MAX   : 100
CALLS : 4
```

If an instrumented piece of code had only been executed once, then the `CYCLES_PRINT` macro would print a message of the form:

```
CYCLES : 95
```

If the switch `-DDO_CYCLE_COUNTS` is not specified, then the macros described above are defined as null macros and no cycle count information is gathered. To switch between development and release mode therefore only requires a re-compilation and will not require any changes to the source of an application.

The macros defined in the `cycles.h` header file may be customized for a particular application without the requirement for rebuilding the run-time libraries.

An example that demonstrates how this facility may be used is:

```
#include <cycles.h>
#include <stdio.h>

extern void foo(void);
extern void bar(void);

extern int
main(void)
{
   cycle_stats_t stats;
   int i;

   CYCLES_INIT(stats)

   for (i = 0; i < LIMIT; i++) {
      CYCLES_START(stats)
      foo();
      CYCLES_STOP(stats)
   }
   printf("Cycles used by foo\n");
   CYCLES_PRINT(stats)
   CYCLES_RESET(stats)

   for (i = 0; i < LIMIT; i++) {
      CYCLES_START(stats)
      bar();
      CYCLES_STOP(stats)
   }
   printf("Cycles used by bar\n");
   CYCLES_PRINT(stats)
}
```

This example might output:

```
Cycles used by foo
   AVG   : 25454
   MIN   : 23003
   MAX   : 26295
   CALLS : 16

Cycles used by bar
   AVG   : 8727
   MIN   : 7653
   MAX   : 8912
   CALLS : 16
```

Alterative methods of measuring the performance of compiled C source are described in the sections "Basic Cycle Counting Facility" on page 3-40 and "Using time.h to Measure Cycle Counts" on page 3-44. Also refer to "Considerations When Measuring Cycle Counts" on page 3-47 which provides some useful tips with regards to performance measurements.

Using time.h to Measure Cycle Counts

The `time.h` header file defines the data type `clock_t`, the `clock` function, and the macro `CLOCKS_PER_SEC`, which together may be used to calculate the number of seconds spent in a program.

In the ANSI C standard, the `clock` function is defined to return the number of implementation dependent clock "ticks" that have elapsed since the program began, and in this version of the C/C++ compiler the function returns the number of processor cycles that an application has used.

The conventional way of using the facilities of the `time.h` header file to measure the time spent in a program is to call the `clock` function at the start of a program, and then subtract this value from the value returned by a subsequent call to the function. This difference is usually cast to a floating-point type, and is then divided by the macro `CLOCKS_PER_SEC` to determine the time in seconds that has occurred between the two calls.

If this method of timing is used by an application then it is important to note that:

- the value assigned to the macro `CLOCKS_PER_SEC` should be independently verified to ensure that it is correct for the particular processor being used (see "Determining the Processor Clock Rate" on page 3-46),
- the result returned by the `clock` function does not include the overhead of calling the library function.

A typical example that demonstrates the use of the `time.h` header file to measure the amount of time that an application takes is shown below.

```
#include <time.h>
#include <stdio.h>

extern int
main(void)
{
    volatile clock_t clock_start;
    volatile clock_t clock_stop;

    double secs;

    clock_start = clock();
    Some_Function_Or_Code_To_Measure();
    clock_stop = clock();

    secs = ((double) (stop_time - start_time))
           / CLOCKS_PER_SEC;
    printf("Time taken is %e seconds\n",secs);
}
```

The header files `cycles.h` and `cycle_count.h` define other methods for benchmarking an application—these header files are described in the sections "Basic Cycle Counting Facility" on page 3-40 and "Cycle Counting Facility with Statistics" on page 3-41, respectively. Also refer to "Considerations When Measuring Cycle Counts" on page 3-47 which provides some guidelines that may be useful.

Determining the Processor Clock Rate

Applications may be benchmarked with respect to how many processor cycles that they use. However, more typically applications are benchmarked with respect to how much time (for example, in seconds) that they take.

To measure the amount of time that an application takes to run on a TigerSHARC processor usually involves first determining the number of cycles that the processor takes, and then dividing this value by the processor's clock rate. The `time.h` header file defines the macro `CLOCKS_PER_SEC` as the number of processor "ticks" per second.

On TigerSHARC architecture, it is set by the run-time library to one of the following values in descending order of precedence:

- via the compile-time switch `-DCLOCKS_PER_SEC=<definition>`.
- via the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- from the `cycles.h` header file

If the value of the macro `CLOCKS_PER_SEC` is taken from the `cycles.h` header file, then be aware that the clock rate of the processor will usually be taken to be the maximum speed of the processor, which is not necessarily the speed of the processor at `RESET`.

Considerations When Measuring Cycle Counts

The run-time library provides three different methods for benchmarking C-compiled code. Each of these alternatives are described in the following sections:

- "Basic Cycle Counting Facility" on page 3-40
  The basic cycle counting facility represents an inexpensive and relatively unobtrusive method for benchmarking C-written source

using cycle counts. The facility is based on macros that factor-in the overhead incurred by the instrumentation. The macros may be customized and they can be switched either or off, and so no source changes are required when moving between development and release mode. The same set of macros is available on other platforms provided by Analog Devices.

- "Cycle Counting Facility with Statistics" on page 3-41
  This is a cycle-counting facility that has more features than the basic cycle counting facility described above. It is therefore more expensive in terms of program memory, data memory, and cycles consumed. However, it does have the ability to record the number of times that the instrumented code has been executed and to calculate the maximum, minimum, and average cost of each iteration. The macros provided take into account the overhead involved in reading the cycle count register. By default, the macros are switched off, but they are switched on by specifying the `-DDO_CYCLE_COUNTS` compile-time switch. The macros may also be customized for a specific application. This cycle counting facility is also available on other Analog Devices architectures.

- "Using time.h to Measure Cycle Counts" on page 3-44
  The facilities of the `time.h` header file represent a simple method for measuring the performance of an application that is portable across a large number of different architectures and systems. These facilities are based around the `clock` function.

  The `clock` function however does not take into account the cost involved in invoking the function. In addition, references to the function may affect the code that the optimizer generates in the vicinity of the function call. This method of benchmarking may not accurately reflect the true cost of the code being measured.

This method is more suited to benchmarking applications rather than smaller sections of code that run for a much shorter time span.

When benchmarking code, some thought is required when adding instrumentation to C source that will be optimized. If the sequence of statements to be measured is not selected carefully, the optimizer may move instructions into (and out of) the code region and/or it may re-site the instrumentation itself, thus leading to distorted measurements. It is therefore generally considered more reliable to measure the cycle count of calling (and returning from) a function rather than a sequence of statements within a function.

It is recommended that variables that are used directly in benchmarking are simple scalars that are allocated in internal memory (be they assigned the result of a reference to the `clock` function, or be they used as arguments to the cycle counting macros). In the case of variables that are assigned the result of the `clock` function, it is also recommended that they be defined with the `volatile` keyword.

The cycle count registers of the TigerSHARC architecture are called the `CCNT0` and `CCNT1` registers. These registers are 32-bit registers. The `CCNT0` register is incremented at every processor cycle; when it wraps back to zero the `CCNT1` register is incremented. Together these registers represent a 64-bit counter that is unlikely to wrap around to zero during the timing of an application.

When running on an ADSP-TS101 architecture, it is possible that the values read from the `CCNT0` and `CCNT1` registers may be affected by bus arbitration issues such as a simultaneous DMA transaction, or by the device being accessed by an external cluster bus master.

File I/O Support

The VisualDSP++ environment provides access to files on a host system by using stdio functions. File I/O support is provided through a set of low-level primitives that implement the open, close, read, write, and seek operations. The functions defined in the `stdio.h` header file make use of these primitives to provide conventional C input and output facilities. The source files for the I/O primitives are available under the TigerSHARC installation of VisualDSP++ in the subdirectory `...\TS\lib\src\libio_src`.

This section describes:

- "Extending I/O Support To New Devices" on page 3-50
- "Default Device Driver Interface" on page 3-58

Refer to "stdio.h" on page 3-17 for information about the conventional C input and output facilities that are provided by the compiler.

Extending I/O Support To New Devices

The I/O primitives are implemented using an extensible device driver mechanism. The default start-up code includes a device driver that can perform I/O through the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Other device drivers may be registered and then used through the normal `stdio` functions.

This section describes:

- "DevEntry Structure"
- "Registering New Devices"
- "Pre-Registering Devices"

- "Default Device"
- "Remove and Rename Functions"

DevEntry Structure

A device driver is a set of primitive functions grouped together into a `DevEntry` structure. This structure is defined in `device.h`.

```
struct DevEntry {
   int   DeviceID;
   void  *data;

   int   (*init)(struct DevEntry *entry);
   int   (*open)(const char *name, int mode);
   int   (*close)(int fd);
   int   (*write)(int fd, unsigned char *buf, int size);
   int   (*read)(int fd, unsigned char *buf, int size);
   long  (*seek)(int fd, long offset, int whence);
   int stdinfd;
   int stdoutfd;
   int stderrfd;
}

typedef struct DevEntry DevEntry;
typedef struct DevEntry *DevEntry_t;
```

The fields within the `DevEntry` structure have the following meanings.

DeviceID:
The `DeviceID` field is a unique identifier for the device, known to the user. Device IDs are used globally across an application.

data:
The `data` field is a pointer for any private data the device may need; it is not used by the run-time libraries.

init:
The `init` field is a pointer to an initialization function. The run-time library calls this function when the device is first registered, passing in the address of this structure (and thus giving the `init` function access to `DeviceID` and the field `data`). If the `init` function encounters an error, it must return -1. Otherwise, it must return a positive value to indicate success.

open:
The `open` field is a pointer to a function performs the "*`open file`*" operation upon the device; the run-time library will call this function in response to requests such as `fopen()`, when the device is the currently-selected default device. The `name` parameter is the path name to the file to be opened, and the `mode` parameter is a bitmask that indicates how the file is to be opened.

```
0x0001   Open file for reading
0x0002   Open file for writing
0x0004   Open file for appending
0x0008   Truncate the file to zero length, if it already exists
0x0010   Create the file, if it does not already exist
```

By default, files are opened as text streams (in which the character sequence `\r\n` is converted to `\n` when reading, and the character `\n` is written to the file as `\r\n`). A file is opened as a binary stream if the following bit value is set in the `mode` parameter:

```
0x0020   Open the file as a binary stream (in raw mode)
```

The `open` function must return a positive "*`file descriptor`*" if it succeeds in opening the file; this file descriptor is used to identify the file to the device in subsequent operations. The file descriptor must be unique for all files currently open for the device, but need not be distinct from file descriptors returned by other devices—the run-time library identifies the file by the combination of device and file descriptor.

If the `open` function fails, it must return -1 to indicate failure.

close:
The `close` field is a pointer to a function that performs the "*`close file`*" operation on the device. The run-time library calls the `close` function in response to requests such as `fclose()` on a stream that was opened on the device. The `fd` parameter is a file descriptor previously returned by a call to the `open` function. The `close` function must return a zero value for success, and a non-zero value for failure.

write:
The `write` field is a pointer to a function that performs the "*`write to file`*" operation on the device. The run-time library calls the `write` function in response to requests, such as `fwrite()`, `fprintf()` and so on, that act on streams that were opened on the device. The `write` function takes three parameters:

- `fd` – this is a file descriptor that identifies the file to be written to; it will be a value that was returned from a previous call to the open function.
- `buf` – a pointer to the data to be written to the file
- `size` – the number of (8-bit) bytes to be written to the file

The `write` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes from `buf` were successfully written to the file
- Zero, indicating that the file has been closed, for some reason (for example, network connection dropped)
- A negative value, indicating an error

read:
The `read` field is a pointer to a function that performs the "*`read from file`*" operation on the device. The run-time library calls the `read` func-

tion in response to requests, such as `fread()`, `fscanf()` and so on, that act on streams that were opened on the device. The `read` function's parameters are:

- `fd` – this is the file descriptor for the file to be read
- `buf` – this is a pointer to the buffer where the retrieved data must be stored
- `size` – this is the number of (8-bit) bytes to read from the file. This must not exceed the space available in the buffer pointed to by `buf`.

The `read` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes were read from the file into `buf`
- Zero, indicating end-of-file
- A negative value, indicating an error

The run-time library expects the `read` function to return `0xa (10)` as the newline character.

seek:
The `seek` field is a pointer to a function that performs dynamic access on the file. The run-time library calls the `seek` function in response to requests such as `rewind()`, `fseek()`, and so on, that act on streams that were opened on the device.

The `seek` function takes the following parameters:

- `fd` – this is the file descriptor for the file which will have its read/write position altered
- `offset` – this is a value that is used to determine the new read/write pointer position within the file; it is in (8-bit) bytes

- `whence` – this is a value that indicates how the `offset` parameter is interpreted:
    - 0: `offset` is an absolute value, giving the new read/write position in the file
    - 1: `offset` is a value relative to the current position within the file
    - 2: `offset` is a value relative to the end of the file

The `seek` function returns a positive value that is the new (absolute) position of the read/write pointer within the file, unless an error is encountered, in which case the `seek` function must return a negative value.

If a device does not support the functionality required by one of these functions (such as read-only devices, or stream devices that do not support seeking), the `DevEntry` structure must still have a pointer to a valid function; the function must arrange to return an error for attempted operations.

stdinfd:
The `stdinfd` field is set to the device file descriptor for `stdin` if the device is expecting to claim the `stdin` stream, or to the enumeration value `dev_not_claimed` otherwise.

stdoutfd:
The `stdoutfd` field is set to the device file descriptor for `stdout` if the device is expecting to claim the `stdout` stream, or to the enumeration value `dev_not_claimed` otherwise.

stderrfd:
The `stderrfd` field is set to the device file descriptor for `stderr` if the device is expecting to claim the `stderr` stream, or to the enumeration value `dev_not_claimed` otherwise.

Registering New Devices

A new device can be registered with the following function:

```
int add_devtab_entry(DevEntry_t entry);
```

If the device is successfully registered, the `init()` routine of the device is called, with `entry` as its parameter. The `add_devtab_entry()` function returns the `DeviceID` of the device registered.

If the device is not successfully registered, a negative value is returned. Reasons for failure include (but are not limited to):

- The `DeviceID` is the same as another device, already registered
- There are no more slots left in the device registry table
- The `DeviceID` is less than zero
- Some of the function pointers are NULL
- The device's `init()` routine returned a failure result
- The device has attempted to claim a standard stream that is already claimed by another device

Pre-Registering Devices

The library source file `devtab.c` (which can be found under a VisualDSP++ installation in the subdirectory `...\TS\lib\src\libio_src`) declares the following array:

```
DevEntry_t DevDrvTable[];
```

This array contains pointers to `DevEntry` structures for each device that is pre-registered, that is, devices that are available as soon as `main()` is entered, and that do not need to be registered at run-time by calling `add_devtab_entry()`. By default, the "*`PrimIO`*" device is registered. The `PrimIO` device provides support for target/host communication when

using the simulators and the Analog Devices emulators and debug agents. This device is pre-registered, so that `printf()` and similar functions operate as expected without additional setup.

Additional devices can be pre-registered by the following process:

1. Take a copy of the `devtab.c` source file and add it to your project.
2. Declare your new device's `DevEntry` structure within the `devtab.c` file, for example,

   ```
   extern DevEntry myDevice;
   ```

3. Include the address of the `DevEntry` structure within the `DevDrvTable[]` array. Ensure that the table is null-terminated. For example,

   ```
   DevEntry_t DevDrvTable[MAXDEV] = {
   #ifdef PRIMIO
      &primio_deventry,
   #endif
      &myDevice,   /* new pre-registered device */
      0,
   };
   ```

All pre-registered devices are initialized by the run-time library when it calls the `init()` function of each of the pre-registered devices in turn.

The normal behavior of the `PrimIO` device when it is registered is to claim the first three files as `stdin`, `stdout` and `stderr`. These standard streams may be re-opened on other devices at run-time by using `freopen()` to close the `PrimIO`-based streams and reopen the streams on the current default device.

To allow an alternative device (either pre-registered or registered by `add_devtab_entry()`) to claim one or all of the standard streams:

1. Take a copy of the `primiolib.c` source file, and add it to your project.

2. Edit the appropriate `stdinfd`, `stdoutfd`, and `stderrfd` file descriptors in the `primio_deventry` structure to have the value `dev_not_claimed`.

3. Ensure the alternative device's `DevEntry` structure has set the standard stream file descriptors appropriately.

Both the device initialization routines, called from the startup code and `add_devtab_entry()`, return with an error if a device attempts to claim a standard stream that is already claimed.

Default Device

Once a device is registered, it can be made the default device using the following function:

```
void set_default_io_device(int);
```

The function should be passed the `DeviceID` of the device. There is a corresponding function for retrieving the current default device:

```
int get_default_io_device(void);
```

The default device is used by `fopen()` when a file is first opened. The `fopen()` function passes the open request to the `open()` function of the device indicated by `get_default_io_device()`. The device's file identifier (`fd`) returned by the `open()` function is private to the device; other devices may simultaneously have other open files that use the same identifier. An open file is uniquely identified by the combination of `DeviceID` and `fd`.

The `fopen()` function records the `DeviceID` and `fd` in the global open file table, and allocates its own internal `fid` to this combination. All future operations on the file use this `fid` to retrieve the `DeviceID` and thus direct the request to the appropriate device's primitive functions, passing the `fd` along with other parameters. Once a file has been opened by `fopen()`, the current value of `get_default_io_device()` is irrelevant to that file.

Remove and Rename Functions

The PrimIO device provides support for the remove() and rename() functions. These functions are not currently part of the extensible File I/O interface, since they deal purely with path names, and not with file descriptors. All calls to remove() and rename() in the run-time library are passed directly to the PrimIO device.

Default Device Driver Interface

The stdio functions provide access to the files on a host system through a device driver that supports a set of low-level I/O primitives. These low-level primitives are described under"Extending I/O Support To New Devices" on page 3-50. The default device driver implements these primitives based on a simple interface provided by the VisualDSP++ simulator and EZ-KIT Lite systems.

All the I/O requests submitted through the default device driver are channeled through the C function _primIO. The assembly label has two underscores, __primIO . The source for this function, and all the other library routines, can be found under the base installation for VisualDSP++ in the subdirectory TS\lib\src\libio_src.

The __primIO function accepts no arguments. Instead, it examines the I/O control block at the label _PrimIOCB. Without external intervention by a host environment, the __primIO routine simply returns, which indicates failure of the request. Two schemes for host interception of I/O requests are provided.

The first scheme is to modify control flow into and out of the __primIO routine. Typically, this would be achieved by a break point mechanism available to a debugger/simulator. Upon entry to __primIO, the data for the request resides in a control block at the label _PrimIOCB. If this scheme is used, the host should arrange to intercept control when it enters the __primIO routine, and, after servicing the request, return control to the calling routine.

The second scheme involves communicating with the DSP process through a pair of simple semaphores. This scheme is most suitable for an externally-hosted development board. Under this scheme, the host system should clear the data word whose label is `__lone_SHARC`; this causes `__primIO` to assume that a host environment is present and able to communicate with the process.

If `__primIO` sees that `__lone_SHARC` is cleared, then upon entry (for example, when an I/O request is made) it sets a non-zero value into the word labeled `__Godot`. The `__primIO` routine then busy-waits until this word is reset to zero by the host. The non-zero value of `__Godot` raised by `__primIO` is the address of the I/O control block.

Data Packing For Primitive I/O

The implementation of the `__primIO` interface is based on a word-addressable machine, with each word comprising a fixed number of 8-bit bytes. All `READ` and `WRITE` requests specify a move of some number of 8-bit bytes, that is, the relevant fields count 8-bit bytes, not words. Packing is always little endian, the first byte of a file read or written is the low-order byte of the first word transferred.

Data packing is set to four bytes per word for the TigerSHARC architecture. Data packing can be changed to accommodate other DSP architectures by modifying the constant `BITS_PER_WORD`, defined in `_wordsize.h`. (For example, a processor with 16-bit addressable words would change this value to 16).

Note that the file name provided in an `OPEN` request uses the processor's "native" string format, normally one byte per word. Data packing applies only to `READ` and `WRITE` requests.

Data Structure for Primitive I/O

The I/O control block is declared in `_primio.h`, as follows.

```
typedef struct
{
   enum
   {
      PRIM_OPEN = 100,
      PRIM_READ,
      PRIM_WRITE,
      PRIM_CLOSE,
      PRIM_SEEK,
      PRIM_REMOVE,
      PRIM_RENAME
   } op;
   int   fileID;
   int   flags;
   unsigned char *buf;   /* data buffer, or file name     */
   int   nDesired;       /* number of characters to read  */
                         /* or write                      */
   int   nCompleted;     /* number of characters actually */
                         /* read or written               */
   void  *more;          /* for future use                */
}
PrimIOCB_T;
```

The first field, `op`, identifies which of the seven currently-supported operations is being requested.

The file ID for an open file is a non-negative integer assigned by the debugger or other "host" mechanism. The `fileID` values 0, 1, and 2 are pre-assigned to `stdin`, `stdout`, and `stderr`, respectively. No open request is required for these file IDs.

Before "activating" the debugger or other host environment, an OPEN or REMOVE request may set the `fileID` field to the length of the filename to open or delete; a RENAME request may also set the field to the length of the old filename. If the `fileID` field does contain a string length, then

this will be indicated in the flags field (see below), and the debugger or other host environment will be able to use the information to perform a batch memory read to extract the filename. If the information is not provided, then the file name has to be extracted one character at a time.

The flags field is a bit field containing other information for special requests. Meaningful bit values for an OPEN operation are:

```
M_OPENR = 0x0001      /* open for reading                    */
M_OPENW = 0x0002      /* open for writing                    */
M_OPENA = 0x0004      /* open for append                     */
M_TRUNCATE = 0x0008   /* truncate to zero length if file exists */
M_CREATE = 0x0010     /* create the file if necessary        */
M_BINARY = 0x0020     /* binary file (vs. text file)         */
M_STRLEN_PROVIDED = 0x8000  /* length of file name(s) available */
```

For a READ operation, the low-order four bits of the flag value contain the number of bytes packed into each word of the read buffer, and the rest of the value is reserved for future use.

For a WRITE operation, the low-order four bits of the flag value contain the number of bytes packed into each word of the write buffer, and the rest of the value form a bit field, for which only the following bit is currently defined:

```
M_ALIGN_BUFFER = 0x10
```

If this bit is set for a WRITE request, the WRITE operation is expected to be aligned on a processor word boundary by writing padding NULs to the file before the buffer contents are transferred.

For an OPEN, REMOVE, and RENAME operation, the debugger (or other host mechanism) has to extract the filename(s) one character at a time from the memory of the target. However, if the bit corresponding to the value M_STRLEN_PROVIDED is set, then the I/O control block contains the length of the filename(s) and the debugger is able to use this information to perform a batch read of the target memory (see the description of the fields fileID and nCompleted).

For a SEEK request, the `flags` field indicates the `seek` mode (`whence`) as:

```
enum
{
   M_SEEK_SET = 0x0001,   /* seek origin is the start of
                             the file                  */
   M_SEEK_CUR = 0x0002,   /* seek origin is the current
                             position within the file  */
   M_SEEK_END = 0x0004,   /* seek origin is the end of
                             the file                  */
};
```

The `flags` field is unused for a CLOSE request.

The `buf` field contains a pointer to the file name for an OPEN or REMOVE request, or a pointer to the data buffer for a READ or WRITE request. For a `RENAME` operation, this field contains a pointer to the old file name.

The `nDesired` field is set to the number of bytes that should be transferred for a READ or WRITE request. This field is also used by a RENAME request, and is set to a pointer to the new file name.

For a SEEK request, the `nDesired` field contains the offset at which the file should be positioned, relative to the origin specified by the `flags` field. (On architectures that only support 16-bit `ints`, the 32-bit offset at which the file should be positioned is stored in the combined fields `[buf, nDesired]`).

The `nCompleted` field is set by `__primIO` to the number of bytes actually transferred by a `READ` or `WRITE` operation. For a `SEEK` operation, `__primIO` sets this field to the new value of the file pointer. (On architectures that only support 16-bit `ints`, `__primIO` sets the new value of the file pointer in the combined fields `[nCompleted, more]`).

The `RENAME` operation may also make use of the `nCompleted` field. If the operation can determine the lengths of the old and new filenames, then it should store these sizes in the fields `fileID` and `nCompleted`, respectively, and also set the bit field `flags` to `M_STRLEN_PROVIDED`. The debugger (or

other host mechanism) can then use this information to perform a batch read of the target memory to extract the filenames. If this information is not provided, then each character of the file names will have to be read individually.

The `more` field is reserved for future use and currently is always set to `NULL` before calling `_primIO`.

Documented Library Functions

The following tables list the documented library functions by the header file in which they are located, whereas "Run-Time Library Reference" on page 3-69 presets the functions in alphabetic order.

Table 3-7. Library Functions in the `complex.h` Header File

arg	cabs	cadd
cartesian	cdiv	cexp
cmlt	conj	csub
norm	polar	

Table 3-8. Library Functions in the `filter.h` Header File

a_compress	a_expand	cfft
cfft2d	cfftf	cfft_mag
convolve	conv2d	fir
fir_decima	fir_interp	ifft
ifft2d	iir	mu_compress
mu_expand	rfft	rfft_mag
rfft2d	rfftf	rfftf_mag
twidfft	twidfftf	

Table 3-9. Library Functions in the `math.h` Header File

acos	alog	alog10
asin	atan	atan2
ceil	cos	cosh
cot	exp	fabs
favg	fclip	floor

Table 3-9. Library Functions in the `math.h` Header File (Cont'd)

fmax	fmin	fmod
frexp	log	log10
modf	pow	rsqrt
sign	sin	sinh
sqrt	tan	tanh

Table 3-10. Library Functions in the `matrix.h` Header File

cmatmadd	cmatmmlt	cmatmsub
cmatsadd	cmatsmlt	cmatssub
matinv	matmadd	matmmlt
matmsub	matsadd	matsmlt
matssub	transpm	

Table 3-11. Library Functions in the `signal.h` Header File

interrupt, interruptf, interrupts, interruptnr, interruptfnr, interruptsnr	raise	signal, signalf, signals, signalnr, signalfnr, signalsnr

Table 3-12. Library Functions in the `stats.h` Header File

autocoh	autocorr	crosscoh
crosscorr	histogramf	mean
rms	var	zero_cross

Table 3-13. Supported Library Functions in `stdio.h` Header File

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fprintf
fputc	fputs	fopen
fread	freopen	fscanf
fseek	fsetpos	ftell
fwrite	getc	getchar
gets	perror	printf
putc	putchar	puts
remove	rename	rewind
scanf	setbuf	setvbuf
snprintf	sprintf	sscanf
ungetc	vfprintf	vprintf
vsnprintf	vsnprintf	

Table 3-14. Library Functions in `stdlib.h` Header File

abs	addbitrev	atof
atoi	atol	atold
atoll	avg	bsearch
clip	count_ones	div
max	min	qsort
rand	srand	strtod
strtof	strtoi	strtol
strtold	strtoll	strtoul
strtoull		

Table 3-15. Library Functions in `time.h` Header File

asctime	clock	ctime
difftime	gmtime	localtime
mktime	strftime	time

Table 3-16. Library Functions in `vector.h` Header File

cvecdot	cvecsadd	cvecsmlt
cvecssub	cvecvadd	cvecvsub
cvecvmlt	vecdot	vecsadd
vecsmlt	vecssub	vecvadd
vecvmlt	vecvsub	

Table 3-17. Library Functions in `window.h` Header File

gen_bartlett	gen_blackman	gen_gaussian
gen_hamming	gen_hanning	gen_harris
gen_kaiser	gen_rectangular	gen_triangle
gen_vonhann		

Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C/C++ programs. This section lists the functions in alphabetical order.

The information that follows applies to all of the functions in the library that are described in this reference.

Notation Conventions
An interval of numbers is indicated by the minimum and maximum, separated by a comma, and enclosed in two square brackets, two parentheses, or one of each. A square bracket indicates that the endpoint is included in the set of numbers; a parenthesis indicates that the endpoint is not included.

Reference Format
Each function in the library has a reference page. These pages have the following format:

Name and Purpose of the function

Synopsis – Required header file and functional prototype

Description – Function specification

Algorithm – High-level mathematical representation of the function

Domain – Range of values supported by the function

Notes – Miscellaneous information

a_compress

A-law compression

Synopsis

```
#include <filter.h>
void a_compress (in, out, n)
const in[];      /* Input array                              */
int out[];       /* Output array                             */
int n;           /* Number of elements to be compressed      */
```

Description

The `a_compress` function takes a vector of linear 13-bit signed speech samples and performs A-law compression according to ITU recommendation G.711. Each sample is compressed to 8 bits and is returned in the vector pointed to by `out`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

C(k)=a-law compression of A(k)

for k=0 to n-1

Domain

content of input array: –4096 to 4095

a_expand

A-law expansion

Synopsis

```
#include <filter.h>
void a_expand (in, out, n)
const int in[];      /* Input array                          */
int out[];           /* Output array                         */
int n;               /* Number of elements to be expanded    */
```

Description

The `a_expand` function inputs a vector of 8-bit compressed speech samples and expands them according to ITU recommendation G.711. Each input value is expanded to a linear 13-bit signed sample in accordance with the A-law definition and is returned in the vector pointed to by `out`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

C(k)=a-law expansion of A(k)

for k=0 to n-1

Domain

Content of input array: 0 to 255

abs

absolute value

Synopsis

```
#include <stdlib.h>
int abs (int x);
long int labs (long int x);
long long int llabs (long long int x);
```

Description

The abs functions return the absolute value of their argument. These functions are built-in functions that cause the compiler to emit an inline instruction to perform the required operation at the point the function is called.

Use "fabs" on page 3-155 to calculate the absolute value of a floating-point number.

Algorithm

Return $|x|$

Domain

Full range for given type.

acos

arc cosine

Synopsis

```
#include <math.h>
double acos (double x);
float acosf (float x);
long double acosd (long double x);
```

Description

The `acos` function returns the arc cosine of the argument. The input must be in the range [–1, 1]. The output, in radians, is in the range [0, π].

Algorithm

$return = cos^{-1}(x)$

Domain

x = [–1.0 ... 1.0]

addbitrev

bit-reversed adder

Synopsis

```
#include <stdlib.h>
int addbitrev (int a,int b);
```

Description

The `addbitrev` function adds the two arguments using the bit-reversed adder. This is binary addition in which the carries are propagated to the right, rather than to the left. Therefore,

```
addbitrev(0x50,0x40) yields 0x30
```

This is useful in algorithms, such as the FFT and interleaver.

The function is a built-in function that causes the compiler to emit an inline instruction to perform the required operation at the point that the function is called.

Algorithm

See “Description”.

alog

anti-log

Synopsis

```
#include <math.h>

float alogf (float x);
double alog (double x);
long double alogd (long double x);
```

Description

The `alog` functions calculate the natural (base `e`) anti-log of their argument. An anti-log function performs the reverse of a `log` function and is therefore equivalent to exponentiation.

The value `HUGE_VAL` is returned if the argument `x` is greater than the function's domain. For input values less than the domain, the functions return 0.0.

Algorithm

$c = e^x$

Domain

`x` = [–87.33 , 88.72] for `alogf()`

`x` = [–708.39 , 709.78] for `alogd()`

Example

```
#include <math.h>

double y;
y = alog(1.0);          /* y = 2.71828... */
```

See Also

alog10, exp, log, pow

alog10

base 10 anti-log

Synopsis

```
#include <math.h>

float alog10f (float x);
double alog10 (double x);
long double alog10d (long double x);
```

Description

The `alog10` functions calculate the base 10 anti-log of their argument. An anti-log function performs the reverse of a `log` function and is therefore equivalent to exponentiation. Therefore, `alog10(x)` is equivalent to `exp(x * log(10.0))`.

The value `HUGE_VAL` is returned if the argument `x` is greater than the function's domain. For input values less than the domain, the functions return 0.0.

Algorithm

$c = e^{(x * \log(10.0))}$

Domain

`x` = [–37.92 , 38.53] for `alog10f()`

`x` = [–307.65 , 308.25] for `alog10d()`

Example

```
#include <math.h>

double y;
y = alog10(1.0);          /* y = 10.0 */
```

See Also

alog, exp, log10, pow

arg

get phase of a complex number

Synopsis

```
#include <complex.h>

float argf (complex_float a);
double arg (complex_double a);
long double argd (complex_long_double a);
```

Description

These functions compute the phase associated with a Cartesian number represented by the complex argument `a`, and return the result.

Algorithm

$$c = atan\left(\frac{\mathrm{Im}(a)}{\mathrm{Re}(a)}\right)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `argf()`

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `argd()`

asctime

convert broken-down time into a string

Synopsis

```
#include <time.h>
char *asctime (const struct tm *t);
```

Description

The `asctime` function converts a broken-down time, as generated by the functions `gmtime` and `localtime`, into an ASCII string that will contain the date and time in the form

```
DDD MMM dd hh:mm:ss YYYY\n
```

where

- `DDD` represents the day of the week (that is. Mon, Tue, Wed, etc.)
- `MMM` is the month and will be of the form Jan, Feb, Mar, etc
- `dd` is the day of the month, from 1 to 31
- `hh` is the number of hours after midnight, from 0 to 23
- `mm` is the minute of the day, from 0 to 59
- `ss` is the second of the day, from 0 to 61 (to allow for leap seconds)
- `YYYY` represents the year

The function returns a pointer to the ASCII string, which may be over-written by a subsequent call to this function. Also note that the function `ctime` returns a string that is identical to

```
asctime(localtime(&t))
```

Error Conditions

The `asctime` function does not return an error condition.

Example

```
#include <time.h>
#include <stdio.h>

struct tm tm_date;

printf("The date is %s",asctime(&tm_date));
```

See Also

ctime, gmtime, localtime

asin

arc sine

Synopsis

```
#include <math.h>
double asin (double x);
float asinf (float x);
long double asind (long double x);
```

Description

The `asin` function returns the arc sine of the argument `x`. The input must be in the range [–1, 1]. The output, in radians, is in the range $-\pi/2$ to $\pi/2$.

Algorithm

return = sin^{-1} (x)

Domain

x = [–1.0 ... 1.0]

atan

arc tangent

Synopsis

```
#include <math.h>
double atan (double x);
float atanf (float x);
long double atand (long double x);
```

Description

The `atan` function returns the arc tangent of the argument. The output, in radians, is in the range -π/2 to π/2.

Algorithm

return = tan^{-1}(x)

Domain

x = [-3.4×10^{38} to $+3.4 \times 10^{38}$] for `atanf()`

x = [-1.7×10^{308} to 1.7×10^{308}] for `atand()`

atan2

arc tangent of quotient

Synopsis

```
#include <math.h>
double atan2 (double y, double x);
float atan2f (float y, float x);
long double atan2d (long double y,long double x);
```

Description

The `atan2` function computes the arc tangent of the input value `y` divided by input value `x`. The output, in radians, is in the range [-π, π].

The function uses the signs of its arguments to compute the quadrant of the return value. Also if `x = 0` and `y > 0`, it returns π/2; and if `x = 0` and `y < 0` , it returns -π/2.

Algorithm

$$return = \tan^{-1}\left(\frac{y}{x}\right)$$

Domain

x,y = [-3.4×10^{38} to $+3.4 \times 10^{38}$] for `atan2f()`

x,y = [-1.7×10^{308} to 1.7×10^{308}] for `atan2d()`

Error Conditions

The `atan2` function returns a zero if y = 0 and x = 0.

atof

convert string to a double

Synopsis

```
#include <stdlib.h>
double atof (const char *nptr)
```

Description

The `atof` function converts a character string to a double value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument may represent either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X` . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan.

Domain

The number should fit within the dynamic range of a `double`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Notes

The function reference `atof (pdata)` is functionally equivalent to:

```
strtod (pdata, (char *) NULL);
```

and therefore, if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of 0.0 or some invalid numerical string.

atoi

convert string to integer

Synopsis

```
#include <stdlib.h>
int atoi (const char *string)
```

Description

The `atoi` function converts a character string to an integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The string argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *`whitespace`* can consist of spaces and/or tab characters, *`sign`* is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `f` for hexadecimal numbers).

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above.

Domain

–2,147,483,648 to 2,147,483,647

atol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long atol (const char *string)
```

Description

The `atol` function converts a character string to a long integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The string argument is as follows:

```
[whitespace][sign][base]digits
```

where *`whitespace`* can consist of spaces and/or tab characters, *`sign`* is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `f` for hexadecimal numbers).

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above.

Domain

–2,147,483,648 to 2,147,483,647

atold

convert string to long double

Synopsis

```
#include <stdlib.h>
long double atold (const char *string)
```

Description

The `atold` function converts a character string to a double-precision floating-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument may represent either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X` . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan.

Domain

The number should fit within the dynamic range of a `long double`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `LDBL_MAX` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Notes

The function reference `atold (pdata)` is functionally equivalent to:

```
strtold (pdata, (char *) NULL);
```

and therefore if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of 0.0 or some invalid numerical string.

atoll

convert string to long long integer

Synopsis

```
#include <stdlib.h>
long long atoll (const char *string)
```

Description

The `atoll` function converts a character string to a long long integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The string argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *`whitespace`* can consist of spaces and/or tab characters, *`sign`* is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `f` for hexadecimal numbers).

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above.

Domain

–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

autocoh

autocoherence

Synopsis

```
#include <stats.h>
void autocohf (a,n,m,c)
const float a[];          /* Input vector a  */
int n;                    /* Input samples   */
int m;                    /* Lag count       */
float c[];                /* Output vector c */
```

Description

The `autocoh` function computes the autocoherence of the input elements contained within input vector `a`, and stores the result to output vector `c`.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

$$c_k = \frac{1}{n} * \sum_{j=0}^{n-k-1} (a_j * a_{j+k}) - (\overline{a})^2$$

where k={0,1,...,m-1} and $\overline{a}$ is the mean value of input vector `a`.

Domain

–3.4 x 10^{38} to +3.4 x 10^{38}

autocorr

autocorrelation

Synopsis

```
#include <stats.h>
void autocorrf (a,n,m,c)
const float a[];            /* Input vector a          */
int n;                      /* Number of input samples */
int m;                      /* Lag count               */
float c[];                  /* Output vector c         */
```

Description

The `autocorr` function computes the autocorrelation of the input elements contained within input vector `a`, and stores the result to output vector `c`. The `autocorr` function is used in digital signal processing applications, such as speech analysis.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} a_j * a_{j+k} \right)$$

where k={0,1,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

avg

mean of two values

Synopsis

```
#include <stdlib.h>

int avg (int a, int b);
long int lavg (long int a, long int b);
long long int llavg (long long int a, long long int b);
```

Description

The `avg` functions add the two arguments and divide the result by two. These functions are built-in functions that cause the compiler to emit an inline instruction to perform the required operation at the point the function is called.

Algorithm

(a + b)/2

Domain

Full range.

bsearch

perform binary search in a sorted array

Synopsis

```
#include <stdlib.h>
void *bsearch (const void *key, const void *base,
               size_t nelem, size_t size,
               int (*compare)(const void *, const void *));
```

Description

The `bsearch` function executes a binary search operation on a pre-sorted array, where

- `key` is a pointer to the element to search for
- `base` points to the start of the array
- `nelem` is the number of elements in the array
- `size` is the size of each element of the array
- `*compare` points to the function used to compare two elements. It takes as parameters a pointer to the key and a pointer to an array element and should return a value less than, equal to, or greater than zero, according to whether the first parameter is less than, equal to, or greater than the second.

The `bsearch` function returns a pointer to the first occurrence of `key` in the array.

Algorithm

Call the compare function with the key and the current node (initially the one in the middle of the array). If the compare returns `0`, then return the current node; if the compare returns `-1`, apply the search recursively on

the portion of the array to the left of the current position; if the compare returns 1, apply the search recursively on the portion of the array to the right of the current location.

Domain

N/A

cabs

complex absolute value

Synopsis

```
#include <complex.h>
float cabsf (a)
complex_float a;      /* Complex input */
```

Description

The `cabs` function computes the complex absolute value of a complex input and returns the result.

Algorithm

$$c = \sqrt{\mathrm{Re}^2(a) + \mathrm{Im}^2(a)}$$

Domain

Re^2 (a) + Im^2 (a) <= 3.4 x 10^{38} for `cabsf( ), cabs( )`

cadd

complex addition

Synopsis

```
#include <complex.h>
complex_float caddf (a,b)
complex_float a;          /* Complex input a */
complex_float b;          /* Complex input b */
```

Description

The `cadd` function adds two complex values `a` and `b`, and returns the result.

Algorithm

$\mathrm{Re}(c) = \mathrm{Re}(a) + \mathrm{Re}(b)$
$\mathrm{Im}(c) = \mathrm{Im}(a) + \mathrm{Im}(b)$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cartesian

convert Cartesian to polar notation

Synopsis

```
#include <complex.h>

float cartesianf (complex_float a, float *phase);
double cartesian (complex_double a, double *phase);
long double cartesiand (complex_long_double a,
                        long double *phase);
```

Description

These functions transform a complex number from Cartesian notation to polar notation. The Cartesian number is represented by the argument `a` that the function converts into a corresponding magnitude, which is returns as the function's result, and a phase that is returned via the second argument `phase`.

Algorithm

```
magnitude = cabs(a)
phase = arg(a)
```

Domain

[-3.4×10^{38} to $+3.4 \times 10^{38}$] for `cartesianf( )`

[-1.7×10^{308} to 1.7×10^{308}] for `cartesiand( )`

Example

```
#include <complex.h>

complex_float point = {-2.0 , 0.0};
float phase;
```

```
float mag;
mag = cartesianf (point,&phase);     /* mag = 2.0, phase = π */
```

cdiv

complex division

Synopsis

```
#include <complex.h>
complex_float cdivf (a,b)
complex_float a;          /* Complex input a */
complex_float b;          /* Complex input b */
```

Description

The `cdiv` function computes the quotient of the division of two complex values and returns the result.

Algorithm

$$\mathrm{Re}(c) = \frac{\mathrm{Re}(a) * \mathrm{Re}(b) + \mathrm{Im}(a) * \mathrm{Im}(b)}{\mathrm{Re}^2(b) + \mathrm{Im}^2(b)}$$

$$\mathrm{Im}(c) = \frac{\mathrm{Re}(b) * \mathrm{Im}(a) - \mathrm{Im}(b) * \mathrm{Re}(a)}{\mathrm{Re}^2(b) + \mathrm{Im}^2(b)}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

ceil

ceiling

Synopsis

```
#include <math.h>
float ceilf (float x)
double ceil (double x)
long double ceild (long double x)
```

Description

The `ceil` function calculates the next highest whole number that is greater than or equal to the floating-point number `x`. It returns the smallest integral value, that is not less than its input.

Algorithm

return = smallest int $\geq$ x

Domain

x = [-3.4×10^{38} to $+3.4 \times 10^{38}$] for `ceilf()`

x = [-1.7×10^{308} to 1.7×10^{308}] for `ceild()`

cexp

complex exponential

Synopsis

```
#include <complex.h>
complex_float cexpf (a)
float a;                /* Value of input */
```

Description

The `cexp` function computes the complex exponential of real input `a` and returns the result.

Algorithm

$\mathrm{Re}(c) = \cos(a)$
$\mathrm{Im}(c) = \sin(a)$

Domain

a = [–1,647,095 ... 1,647,095] for `cexpf()`
a = [–843,314,850 ... 843,314,850] for `cexpd()`

cfft

N point complex input FFT

Synopsis

```
#include <filter.h>

void cfft (in[], t[], out[], w[], wst, n)
const complex_float in[];         /* Input sequence           */
complex_float t[];                /* Temporary working buffer */
complex_float out[];              /* Output sequence          */
const complex_float w[];          /* Twiddle sequence         */
int wst;                          /* Twiddle factor stride    */
int n;                            /* Number of FFT points     */
```

Description

The `cfft` function transforms the time domain complex input signal sequence to the frequency domain by using the accelerated version of the 'Discrete Fourier Transformation' known as a 'Fast Fourier Transform' or FFT. The `cfft` function "decimates in frequency" by the best choice FFT algorithm, radix-4 or mixed radix, depending on the input sequence length.

The size of the input array `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` represents the number of points in the FFT. If the input data can be overwritten, then the memory requirements may be reduced by specifying the input array as the output array. Run-time performance of the function is improved if the input and output arrays are allocated in a different memory block than the twiddle table, `w`.

The twiddle table is passed in the argument `w`, which must contain at least ¾`n` complex twiddle factors. The function `twidfft` may be used to initialize the array. If the twiddle table contains more factors than needed for a

particular call on `cfft`, then the stride factor has to be set appropriately; otherwise it should be 1. Refer to "twidfft" on page 3-336 for more information.

The library also contains the `cfftf` function (see on page 3-111), which is an optimized implementation of a complex FFT using a fast radix-2 algorithm. The `cfftf` function however imposes certain memory alignment requirements that may not be appropriate for some applications.

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

When the sequence length, `n`, is a power of four, the radix-4 method is used. When the sequence length is a power of two (see "Domain"), the mixed radix method is used. At the first stage of the decimation in frequency, the `cfft` function uses the radix-2 method to generate two sequences with `n/2` points each. The `n/2` is now a power of four which creates the condition to employ the faster radix-4 method over these two sequences.

Domain

Input sequence length `n` must be equal to either a power of two, or a power of four, and at least 16.

Example

```
/* Example to demonstrate how to generate two complex FFTs using
   a single twiddle table */

#include <filter.h>

#define NDATA1 256
```

```
#define NDATA2 32

complex_float data1[NDATA1];   /* data for a 256-point FFT */
complex_float data2[NDATA2];   /* data for a 32-point FFT  */

complex_float output1[NDATA1];
complex_float output2[NDATA2];

static complex_float twidtab[(3*NDATA1)/4];
complex_float temp[NDATA1];
     /* note that the temporary buffer should be as large as
         the largest FFT generated */

/* Generate a twiddle table for a 256-point FFT */

twidfft (twidtab,NDATA1);
     /* note that a twiddle table is constant for a given number
         of FFT points */

/* Generate a 256-point complex FFT */

cfft (data1, temp, output1, twidtab, 1, NDATA1);
     /* note that the twiddle table stride factor is 1 */

/* Generate a 32-point complex FFT */

cfft (data2, temp, output2, twidtab, (NDATA1/NDATA2), NDATA2);
     /* note that the twiddle table stride factor is 8 */
```

cfft_mag

cfft magnitude

Synopsis

```
#include <filter.h>

void cfft_mag (const complex_float input[],
               float output[],
               int fftsize);
```

Description

The `cfft_mag` function computes a normalized power spectrum from the output signal generated by a `cfft` or `cfftf` function. The size of the signal and the size of the power spectrum is `fftsize`.

The Nyquist frequency is located at `(fftsize/2) + 1`.

Algorithm

$$\text{magnitude}(z) = \frac{\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

Example

```
#include <filter.h>
#define N 64

complex_float fft_input[N];
complex_float fft_output[N];

complex_float temp[N];
complex_float twid[(3*N)/4];

float spectrum[N];
```

```
/* Generate a twiddle table for the FFT */

twidfft (twid,N);

    /* Note that a twiddle table is constant for a given number
       of FFT points */

/* Generate a complex FFT using cfft */

cfft (fft_input, temp, fft_output, twid, 1, N);

/* Generate the power spectrum */

cfft_mag (fft_output, spectrum, N);
```

cfft2d

NxN point 2-D complex input FFT

Synopsis

```
#include <filter.h>
void cfft2d (*in, *t, *out, w[], wst, n)
const complex_float *in;  /* Pointer to input matrix a[n][n]   */
complex_float *t;         /* Pointer to working buffer t[n][n] */
complex_float *out;       /* Pointer to matrix c[n][n]         */
const complex_float w[];  /* Twiddle sequence                  */
int wst;                  /* Twiddle factor stride             */
int n;                    /* Number of FFT points              */
```

Description

The `cfft2d` function computes the two-dimensional Fast Fourier Transform of the complex input matrix `a[n][n]`, and stores the result to the complex matrix `c[n][n]`.

If the input data can be overwritten, optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `cfft2d`, then the stride parameter needs to be set appropriately; otherwise, it should be one. For more information, see “twidfft” on page 3-336.

Algorithm

$$c(i,j) = \sum_{k=0}^{n-1}\sum_{l=0}^{n-1} a(k,l) * e^{-2\pi j(i*k+j*l)/n}$$

where i={0,1,...,n -1}, j={0,1,2,...,n-1}

Domain

Input sequence length `n` must be equal to either a power of two, or a power of four, and at least 16.

cfftf

fast N point complex input FFT

Synopsis

```
#include <filter.h>
void cfftf (in[], out[], twid[], wst, n)
const complex_float in[];      /* Input sequence         */
complex_float out[];           /* Output sequence        */
const complex_float twid[];    /* Twiddle sequence       */
int wst;                       /* Twiddle factor stride  */
int n;                         /* Number of FFT points   */
```

Description

The `cfftf` function transforms the time domain complex input signal sequence to the frequency domain by using the accelerated version of the 'Discrete Fourier Transform' known as a 'Fast Fourier Transform' or FFT. It "decimates in frequency" using an optimized radix-2 algorithm.

The size of the input array `in` and the output array `out` is `n`, where `n` represents the number of points in the FFT. The `cfftf` function has been designed for optimum performance and requires that the input array `in` be aligned on an address boundary that is a multiple of twice the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the `cfft` function instead with no loss of facility (apart from performance).

The twiddle table is passed in the argument `twid`, which must contain at least `n/2` complex twiddle factors. The function `twidfftf` may be used to initialize the array. If the twiddle table contains more factors than required for a particular FFT size, then the stride factor `wst` has to be set appropriately; otherwise, it should be set to 1. For more information, see "twidfftf" on page 3-338.

The twiddle tables used by the functions `cfft` and `cfftf` are not compatible. The `cfft` function (see on page 3-104) uses a twiddle table that contains ¾`n` factors in which the imaginary coefficients are positive sine values, while the `cfftf` function uses a twiddle table with ½`n` factors in which the imaginary coefficients are negative sine values.

It is recommended that the twiddle table and the output array are allocated in separate memory blocks—otherwise, the performance of the function degrades.

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

Domain

The number of points in the FFT must be a power of 2 and must be at least 32.

clearerr

clear file or stream error indicator

Synopsis

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

The `clearerr` function clears the error and end-of-file (`EOF`) indicators for the particular stream pointed to by `stream`.

The `stream` error indicators record whether any read or write errors have occurred on the associated stream. The `EOF` indicator records when there is no more data in the file.

Error Conditions

The `clearerr` function does not return an error condition.

Example

```
#include <stdio.h>
FILE *routine(char *filename)
{
   FILE *fp;
   fp = fopen(filename, "r");
   /* Some operations using the file */
   /* now clear the error indicators for the stream */
   clearerr(fp);
   return fp;
}
```

See Also

feof, ferror

clip

clip

Synopsis

```
#include <stdlib.h>

int clip (int parm1, int parm2);
long int lclip (long int parm1, long int parm2);
long long int llclip (long long int parm1, long long int parm2);
```

Description

The clip functions return their first argument if it is less than the absolute value of the second argument, otherwise they return the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative. The clip functions are built-in functions which are implemented with a `CLIP` instruction.

Algorithm

```
if ( |parm1| < |parm2| )
   return( parm1 )
else
   return( |parm2| * signof(parm1))
```

Domain

Full range.

clock

processor time

Synopsis

```
#include <time.h>
clock_t clock (void);
```

Description

The `clock` function returns the number of processor cycles that have elapsed since an arbitrary starting point. The function returns the value `(clock_t) -1`, if the processor time is not available or if it cannot be represented. The result returned by the function may be used to calculate the processor time in seconds by dividing it by the macro `CLOCKS_PER_SEC`. For more information, see "time.h" on page 3-21. An alternative method of measuring the performance of an application is described in "Measuring Cycle Counts" on page 3-39.

Error Conditions

The `clock` function does not return an error condition.

Example

```
#include <time.h>

time_t start_time,stop_time;
double time_used;

start_time = clock();
compute();
stop_time = clock();

time_used = ((double) (stop_time - start_time)) / CLOCKS_PER_SEC;
```

See Also

No references to this function.

cmatmadd

complex matrix + matrix addition

Synopsis

```
#include <matrix.h>
void cmatmaddf (a,b,n,m,c)
const complex_float *a;   /* Pointer to input matrix a[][]     */
const complex_float *b;   /* Pointer to input matrix b[][]     */
int n;                    /* Number of rows in matrix a[][]    */
int m;                    /* Number of columns in matrix a[][] */
complex_float *c;         /* Pointer to matrix c[][]           */
```

Description

This function computes the addition of input complex matrix `a[][]` with input complex matrix `b[][]`, and stores the result to output complex matrix `c[][]`. The dimensions of complex matrix `a[][]` are `n` and `m` and the dimensions of complex matrix `b` are `n` and `m`. The resulting output complex matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$\mathrm{Re}(c_{i,j}) = \mathrm{Re}(a_{i,j}) + \mathrm{Re}(b_{i,j})$$
$$\mathrm{Im}(c_{i,j}) = \mathrm{Im}(a_{i,j}) + \mathrm{Im}(b_{i,j})$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatmmlt

complex matrix * matrix multiplication

Synopsis

```
#include <matrix.h>
void cmatmmltf (a,n,k,b,m,c)
const complex_float *a;   /* Pointer to input matrix a[][]     */
int n;                    /* Number of rows in matrix a[][]    */
int k;                    /* Number of columns in matrix a[][] */
const complex_float *b;   /* Pointer to input matrix b[][]     */
int m;                    /* Number of columns in matrix b[][] */
complex_float *c;         /* Pointer to matrix c[][]           */
```

Description

This function computes the multiplication of input complex matrix `a[][]` with input complex matrix `b[][]`, and stores the result to output complex matrix `c[][]`. The dimensions of complex matrix `a[][]` are `n` and `k` and the dimensions of complex matrix `b` are `k` and `m`. The resulting output complex matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$\mathrm{Re}(c_{i,j}) = \sum_{l=0}^{k-1} (\mathrm{Re}(a_{i,l}) * \mathrm{Re}(b_{l,j}) - \mathrm{Im}(a_{i,l}) * \mathrm{Im}(b_{l,j}))$$

$$\mathrm{Im}(c_{i,j}) = \sum_{l=0}^{k-1} (\mathrm{Re}(a_{i,l}) * \mathrm{Im}(b_{l,j}) + \mathrm{Im}(a_{i,l}) * \mathrm{Re}(b_{l,j}))$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4 x 10^{38} to +3.4 x 10^{38}

cmatmsub

complex matrix - matrix subtraction

Synopsis

```
#include <matrix.h>
void cmatmsubf (a,b,n,m,c)
const complex_float *a;   /* Pointer to input matrix a[][]     */
const complex_float *b;   /* Pointer to input matrix b[][]     */
int n;                    /* Number of rows in matrix a[][]    */
int m;                    /* Number of columns in matrix a[][] */
complex_float *c;         /* Pointer to matrix c[][]           */
```

Description

This function computes the subtraction of input complex matrix `a[][]` with input complex matrix `b[][]`, and stores the result to output complex matrix `c[][]`. The dimensions of complex matrix `a[][]` are `n` and `m` and the dimensions of complex matrix `b` are `n` and `m`. The resulting output complex matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$\mathrm{Re}(c_{i,j}) = \mathrm{Re}(a_{i,j}) - \mathrm{Re}(b_{i,j})$$
$$\mathrm{Im}(c_{i,j}) = \mathrm{Im}(a_{i,j}) - \mathrm{Im}(b_{i,j})$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatsadd

complex matrix + scalar addition

Synopsis

```
#include <matrix.h>
void cmatsaddf (a,b,n,m,c)
const complex_float *a;   /* Pointer to input matrix a[][]     */
const complex_float b;    /* Input scalar b                    */
int n;                    /* Number of rows in matrix a[][]    */
int m;                    /* Number of columns in matrix a[][] */
complex_float *c;         /* Pointer to matrix c[][]           */
```

Description

This function adds the complex scalar value `b` to each element of complex matrix `a[][]`, placing the result in complex matrix `c[][]`. The dimensions of complex matrix `a[][]` are `n` and `m`. The resulting output complex matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$\mathrm{Re}(c_{i,j}) = \mathrm{Re}(a_{i,j}) + \mathrm{Re}(b)$$
$$\mathrm{Im}(c_{i,j}) = \mathrm{Im}(a_{i,j}) + \mathrm{Im}(b)$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatsmlt

complex matrix * scalar multiplication

Synopsis

```
#include <matrix.h>
void cmatsmltf (a,b,n,m,c)
const complex_float *a;   /* Pointer to input matrix a[][]     */
const complex_float b;    /* Input scalar b                    */
int n;                    /* Number of rows in matrix a[][]    */
int m;                    /* Number of columns in matrix a[][] */
complex_float *c;         /* Pointer to matrix c[][]           */
```

Description

This function computes the multiplication of input complex matrix `a[][]` with input complex scalar `b`, and stores the result to output complex matrix `c[][]`. The dimensions of complex matrix `a[][]` are `n` and `m`. The resulting output complex matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$\mathrm{Re}(c_{i,j}) = \mathrm{Re}(a_{i,j}) * \mathrm{Re}(b) - \mathrm{Im}(a_{i,j}) * \mathrm{Im}(b)$$
$$\mathrm{Im}(c_{i,j}) = \mathrm{Re}(a_{i,j}) * \mathrm{Im}(b) + \mathrm{Im}(a_{i,j}) * \mathrm{Re}(b)$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatssub

complex matrix - scalar subtraction

Synopsis

```
#include <matrix.h>
void cmatssubf (a,b,n,m,c)
const complex_float *a;   /* Pointer to input matrix a[][]     */
const complex_float b;    /* Input scalar b                    */
int n;                    /* Number of rows in matrix a[][]    */
int m;                    /* Number of columns in matrix a[][] */
complex_float *c;         /* Pointer to matrix c[][]           */
```

Description

This function computes the subtraction of input complex matrix `a[][]` with input complex scalar `b`, and stores the result to output complex matrix `c[][]`. The dimensions of complex matrix `a[][]` are `n` and `m`. The resulting output complex matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$\mathrm{Re}(c_{i,j}) = \mathrm{Re}(a_{i,j}) - \mathrm{Re}(b)$$
$$\mathrm{Im}(c_{i,j}) = \mathrm{Im}(a_{i,j}) - \mathrm{Im}(b)$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmlt

complex multiply

Synopsis

```
#include <complex.h>
complex_float cmltf (a,b)
complex_float a;          /* Complex input a */
complex_float b;          /* Complex input b */
```

Description

This function multiplies two complex values `a` and `b`, and returns the result.

Algorithm

$$\mathrm{Re}(c) = \mathrm{Re}(a) * \mathrm{Re}(b) - \mathrm{Im}(a) * \mathrm{Im}(b)$$
$$\mathrm{Im}(c) = \mathrm{Re}(a) * \mathrm{Im}(b) + \mathrm{Im}(a) * \mathrm{Re}(b)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

conj

complex conjugate

Synopsis

```
#include <complex.h>
complex_float conjf (a)
complex_float a;      /* Complex input a */
```

Description

This function conjugates the complex input `a`, and returns the result.

Algorithm

$\mathrm{Re}(c) = \mathrm{Re}(a)$

$\mathrm{Im}(c) = -\mathrm{Im}(a)$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

convolve

convolution

Synopsis

```
#include <filter.h>
void convolve (vin1, vlen1, vin2, vlen2, vout)
const float vin1[];         /* Pointer to input sequence 1    */
int vlen1;                  /* Length of the input sequence 1 */
const float vin2[];         /* Pointer to input sequence 2    */
int vlen2;                  /* Length of the input sequence 2 */
float vout[];               /* Pointer to output sequence     */
```

Description

This function convolves two sequences pointed to by `vin1` and `vin2`. If `vin1` points to the sequence whose length is `vlen1` and `vin2` points to the sequence whose length is `vlen2`, the resulting sequence pointed to by `vout` has the length `vlen1 + vlen2 - 1`.

Algorithm

Convolution between two sequences `cin1` and `cin2` is described as:

$$cout(n) = \sum_{k=0}^{k=vlen2-1} cin1(n+k) \bullet cin2(vlen2-k-1)$$

for n = {0,1,2,...,vlen-1)

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

conv2d

2-D convolution

Synopsis

```
#include <filter.h>
void conv2d (min1, mrow1, mcol1, min2, mrow2, mcol2, mout )
const float *min1;          /* Pointer to input matrix 1     */
int mrow1;                  /* Number of rows in matrix 1    */
int mcol1;                  /* Number of columns in matrix 1 */
const float *min2;          /* Pointer to input matrix 2     */
int mrow2;                  /* Number of rows in matrix 1    */
int mcol2;                  /* Number of columns in matrix 2 */
int *mout;                  /* Pointer to matrix             */
```

Description

This function computes the two-dimensional convolution of input matrix `min1` of size `mrow1` x `mcol1` and `min2` of size `mrow2` x `mcol2` and stores the result in matrix `mout` of dimension (`mrow1` +`mrow2-1`) x (`mcol1` + `mcol2-1`).

Algorithm

Two-dimensional input matrix `min1` is convolved with input matrix `min2`, placing the result in a matrix pointed to by `mout`.

$$mout(r,c) = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} \min 2[k-1-i][k-1-j] \bullet \min 1[r+i][c+j]$$

for r=0 to nr-k+1 and for c=0 to nc-k+1

Domain

–3.4 x 10^{38} to +3.4 x 10^{38}

copysign

copy the sign

Synopsis

```
#include <math.h>
float copysignf (float parm1, float parm2)
double copysign (double parm1, double parm2)
long double copysignd (long double parm1, long double parm2)
```

Description

This function copies the sign of the second argument to the first argument.

Algorithm

```
return (|parm1| * copysignof(parm2))
```

Domain

Full range for type of parameters used

cos

cosine

Synopsis

```
#include <math.h>
float cosf (float x)
double cos (double x)
long double cosd (long double x)
```

Description

This function calculates the cosine of number `x` where `x` is measured in radians. The output is in the range [-1 to 1]. If `x` is outside of the domain, this function returns 0.0.

Algorithm

```
return = cos(x)
```

Domain

x = [–1,647,095 ... 1,647,095] for `cosf()`

x = [–843,314,850 ... 843,314,850] for `cosd()`

cosh

hyperbolic cosine

Synopsis

```
#include <math.h>
float coshf (float x)
double cosh (double x)
long double coshd (long double x)
```

Description

This function calculates the hyperbolic cosine of a number `x` where `x` is measured in radians. If `x` is outside the domain, this function returns 3.4×10^{38} for a `float`-type return value and 1.7×10^{308} for a `double`-type return value.

Algorithm

```
return = cosh(x)
```

Domain

$x = [-(\ln(3.4 \times 10^{38} - \ln(2)) \ldots (\ln(3.4 \times 10^{38}) - \ln(2))]$ for `coshf()`

$x = [-(\ln(1.7 \times 10^{308}) - \ln(2)) \ldots (\ln(1.7 \times 10^{308}) - \ln(2))]$ for `coshd()`

cot

cotangent

Synopsis

```
#include <math.h>
float cotf (float x)
double cot (double x)
long double cotd (long double x)
```

Description

This function calculates the cotangent of number `x` where `x` is measured in radians. If `x` is outside of the domain, this function returns 0.0.

Algorithm

```
return = cot(x)
```

Domain

x = [–6,588,397 ... 6,588,397] for `cotf()`
x = [–421,657,424 ... 421,657,424] for `cotd()`

count_ones

count one bits in word

Synopsis

```
#include <stdlib.h>
int count_ones (int parm)
int lcount_ones (long parm)
int llcount_ones (long long parm)
```

Description

These functions count the number of one bits in the argument `parm`.

Algorithm

$$return = \sum_{j=0}^{j=N-1} bit[\,j\,]\ of\ parm$$

where `N` is the number of bits in `parm`.

crosscoh

cross-coherence

Synopsis

```
#include <stats.h>
void crosscohf (a,b,n,m,c)
const float a[];          /* Input vector a          */
const float b[];          /* Input vector b          */
int n;                    /* Number of input samples */
int m;                    /* Lag count               */
float c[];                /* Output vector c         */
```

Description

This function computes the cross-coherence of the input elements contained within input vector `a` and input vector `b` and stores the result in the output vector `c`.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} (a_j - \bar{a}) * (b_{j+k} - a) \right)$$

where k={0,1,...,m-1}, a is the mean value of input vector a and is b the mean value of input vector b.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

crosscorr

cross-correlation

Synopsis

```
#include <stats.h>
void crosscorrf (a,b,n,m,c)
const float a[];          /* Input vector a              */
const float b[];          /* Input vector b              */
int n;                    /* Number of input samples     */
int m;                    /* Lag count                   */
float c[];                /* Pointer to output vector c  */
```

Description

This function computes the cross-correlation of the input elements contained within input vector `a` and input vector `b` and places the result in the output vector `c`.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} a_j * b_{j+k} \right)$$

where k={0,1,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

csub

complex subtraction

Synopsis

```
#include <complex.h>
complex_float csubf (a,b)
complex_float a;      /* Complex input a */
complex_float b;      /* Complex input b */
```

Description

This function computes the complex subtraction of two complex inputs `a` and `b` and returns the result.

Algorithm

$\mathrm{Re}(c) = \mathrm{Re}(a) - \mathrm{Re}(b)$

$\mathrm{Im}(c) = \mathrm{Im}(a) - \mathrm{Im}(b)$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

ctime

convert calendar time into a string

Synopsis

```
#include <time.h>
char *ctime (const time_t *t);
```

Description

The `ctime` function converts a calendar time, pointed to by the argument `t` into a string that represents the local date and time. The form of the string is the same as that generated by `asctime`, and so a call to `ctime` is equivalent to

```
asctime(localtime(&t))
```

A pointer to the string is returned by `ctime`, and it may be overwritten by a subsequent call to the function.

Error Conditions

The `ctime` function does not return an error condition.

Example

```
#include <time.h>
#include <stdio.h>

time_t cal_time;

if (cal_time != (time_t)-1)
    printf("Date and Time is %s",ctime(&cal_time));
```

See Also

asctime, gmtime, localtime, time

cvecdot

complex vector dot product

Synopsis

```
#include <vector.h>
complex_float cvecdotf (a,b,n)
const complex_float a[];      /* Input vector a  */
const complex_float b[];      /* Input vector b  */
int n;                        /* Element count   */
```

Description

This function computes the complex dot product of two complex input vectors `a` and `b` and returns the complex result.

Algorithm

The algorithm for a complex dot product is given by:

$$\mathrm{Re}(c_i) = \sum_{l=0}^{n-1} \mathrm{Re}(a_i) * \mathrm{Re}(b_i) - \mathrm{Im}(a_i) * \mathrm{Im}(b_i)$$

$$\mathrm{Im}(c_i) = \sum_{l=0}^{n-1} \mathrm{Re}(a_i) * \mathrm{Im}(b_i) + \mathrm{Im}(a_i) * \mathrm{Re}(b_i)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecsadd

complex vector + scalar addition

Synopsis

```
#include <vector.h>
void cvecsaddf (a,b,c,n)
const complex_float a[];          /* Input vector a */
complex_float b;                  /* Input scalar b */
complex_float c[];                /* Output vector  */
int n;                            /* Element count  */
```

Description

This function adds input complex scalar `b` to each element of input complex vector `a` and stores the results in the output complex vector `c`.

Algorithm

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i) + \mathrm{Re}(b)$$
$$\mathrm{Im}(c_i) = \mathrm{Im}(a_i) + \mathrm{Im}(b)$$

where i={0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecsmlt

complex vector * scalar multiplication

Synopsis

```
#include <vector.h>
void cvecsmltf (a,b,c,n)
const complex_float a[];         /* Input vector a */
complex_float b;                 /* Input scalar b */
complex_float c[];               /* Output vector  */
int n;                           /* Element count  */
```

Description

This function multiplies each element of input complex vector `a` by input complex scalar `b` and stores the results in the output complex vector `c`.

Algorithm

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(b) - \mathrm{Im}(a_i)*\mathrm{Im}(b)$$
$$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(b) + \mathrm{Im}(a_i)*\mathrm{Re}(b)$$

where i={0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecssub

complex vector - scalar subtraction

Synopsis

```
#include <vector.h>
void cvecssubf (a,b,c,n)
const complex_float a[];         /* Input vector a */
complex_float b;                 /* Input scalar b */
complex_float c[];               /* Output vector  */
int n;                           /* Element count  */
```

Description

This function subtracts input complex scalar `b` from each element of input complex vector `a` and stores the results in the output complex vector `c`.

Algorithm

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i) - \mathrm{Re}(b)$$
$$\mathrm{Im}(c_i) = \mathrm{Im}(a_i) - \mathrm{Im}(b)$$

where i={0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecvadd

complex vector + vector addition

Synopsis

```
#include <vector.h>
void cvecvaddf (a,b,c,n)
const complex_float a[];         /* Input vector a */
const complex_float b[];         /* Input vector b */
complex_float c[];               /* Output vector  */
int n;                           /* Element count  */
```

Description

This function adds two input vectors and stores the results in the output vector `c`.

Algorithm

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i) + \mathrm{Re}(b_i)$$
$$\mathrm{Im}(c_i) = \mathrm{Im}(a_i) + \mathrm{Im}(b_i)$$

where i={0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecvmlt

complex vector * vector multiplication

Synopsis

```
#include <vector.h>
void cvecvmltf (a,b,c,n)
const complex_float a[];          /* Input vector a */
const complex_float b[];          /* Input vector b */
complex_float c[];                /* Output vector  */
int n;                            /* Element count  */
```

Description

This function multiplies two input vectors `a` and `b` and stores the results in the output vector `c`.

Algorithm

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(b_i) - \mathrm{Im}(a_i)*\mathrm{Im}(b_i)$$
$$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(b_i) + \mathrm{Im}(a_i)*\mathrm{Re}(b_i)$$

where i={0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecvsub

complex vector-to-vector subtraction

Synopsis

```
#include <vector.h>
void cvecvsubf (a,b,c,n)
const complex_float a[];         /* Input vector a */
const complex_float b[];         /* Input vector b */
complex_float c[];               /* Output vector  */
int n;                           /* Element count  */
```

Description

This function subtracts input vector `b` from input vector `a` and stores the results in the output vector `c`.

Algorithm

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i) - \mathrm{Re}(b_i)$$
$$\mathrm{Im}(c_i) = \mathrm{Im}(a_i) - \mathrm{Im}(b_i)$$

where i={0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

difftime

difference between two calendar times

Synopsis

```
#include <time.h>
double difftime (time_t t1, time_t t0);
```

Description

The `difftime` function returns the difference in seconds between two calendar times, expressed as a `double`. By default, the `double` data type represents a 32-bit, single precision, floating-point, value. This form is normally insufficient to preserve all of the bits associated with the difference between two calendar times, particularly if the difference represents more than 97 days. It is recommended therefore that any function that calls `difftime` is compiled with the `-double-size-64` switch.

Error Conditions

The `difftime` function does not return an error condition.

Example

```
#include <time.h>
#include <stdio.h>
#define NA ((time_t)(-1))

time_t cal_time1;
time_t cal_time2;
double time_diff;

if ((cal_time1 == NA) || (cal_time2 == NA))
    printf("calendar time difference is not available\n");
else
    time_diff = difftime(cal_time2,cal_time1);
```

See Also

time

div

division

Synopsis

```
#include <stdlib.h>
div_t div (int numer, int denom);
```

Description

The `div` function divides `numer` by `denom`, both of type `int`, and returns a structure of type `div_t`. The type `div_t` is defined as

```
typedef struct {
   int quot;
   int rem;
} div_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if `result` is of type `div_t`,

```
result.quot * denom + result.rem == numer
```

Algorithm

$$Quotient = \left\lfloor \frac{|(numer)|}{|denom|} \right\rfloor \times signof\left(\frac{(numer)}{denom}\right)$$

Remainder = numer - (Quotient x denom)

Domain

numerator: –2,147,483,648 to 2,147,483,647
denominator: –2,147,483,648 to –1 and 1 to 2,147,483,647

exp

exponential

Synopsis

```
#include <math.h>

float expf (float x)
double exp (double x)
long double expd (long double x)
```

Description

The `exp` function computes the exponential value `e` to the power of its argument.

The value `HUGE_VAL` is returned if the argument `x` is greater than the function's domain. For input values less than the domain, the functions return 0.0.

Algorithm

return = $e^{(x)}$

Domain

x = [–87.33 , 88.72] for `expf()`

x = [–708.39 , 709.78] for `expd()`

__emuclk

Get simulator cycle count

Synopsis

```
#include <libsim.h>
int __emuclk (void)
```

Description

This function returns the current value of the simulator cycle count. Note that the function name has two leading underscores.

The facilities defined in the `libsim.h` header file are included in the `libsim.dlb` run-time library, which is supported only under the Analog Devices simulator. The default `.LDF` file will add the library to the list of libraries to be searched by the linker but only if the `-flags-link -MD__USING_LIBSIM=1` linker command-line switch is specified when building an application.

The `libsim.h` functionality is not available in byte-addressing mode.

Algorithm

See “Description”.

fabs

float absolute value

Synopsis

```
#include <math.h>

double fabs (double f);
float fabsf (float f);
long double fabsd (long double f);
```

Description

The `fabs` functions return the absolute value of their argument.

The `fabsf` function is a built-in function which is implemented with a `ABS` instruction; the `fabs` function is compiled as a built-in function if `double` is the same size as `float`.

Algorithm

return $|x|$

Example

```
#include <math.h>
double y;

y = fabs(-2.3);       /* y = 2.3 */
y = fabs(2.3);        /* y = 2.3 */
```

See Also

abs

favg

mean of two values

Synopsis

```
#include <math.h>

float favgf (float a, float b);
double favg (double a, double b);
long double favgd (long double a, long double b);
```

Description

The `favg` functions add the two arguments and divide the result by two.

The `favgf` function is a built-in function that causes the compiler to emit an inline instruction to perform the required operation at the point the function is called; the `favg` function is compiled as a built-in function if `double` is the same size as `float`.

Algorithm

```
(a + b) / 2
```

Domain

Full range.

fclip

clip `x` by `y`

Synopsis

```
#include <math.h>

float fclipf (float parm1, float parm2);
double fclip (double parm1, double parm2);
long double fclipd (long double parm1, long double parm2);
```

Description

The `fclip` functions return their first argument if it is less than the absolute value of the second argument; otherwise, they return the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative.

The `fclipf` function is a built-in function which is implemented with a `CLIP` instruction; the `fclip` function is compiled as a built-in function if `double` is the same size as `float`.

Algorithm

```
if ( |parm1| < |parm2| )
    return (parm1)
else
    return (|parm2| * signof(parm1))
```

Domain

Full range.

fclose

close a stream

Synopsis

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

The `fclose` function flushes `stream` and closes the associated file. The flush will result in any unwritten buffered data for the stream to be written to the file, with any unread buffered data being discarded.

If the buffer associated with `stream` was allocated automatically, it will be deallocated.

The `fclose` function will return zero on successful completion.

Error Conditions

If the `fclose` function is unsuccessful, it returns `EOF`.

Example

```
#include <stdio.h>
void example(char* fname)
{
   FILE *fp;
   fp = fopen(fname, "w+");
   /* Do some operations on the file */
   fclose(fp);
}
```

See Also

fopen

feof

test for end of file

Synopsis

```
#include <stdio.h>
int feof(FILE *stream);
```

Description

The `feof` function tests whether or not the file identified by `stream` has reached the end of the file. The routine returns 0 if the end of the file has not been reached and a non-zero result of the end of file has been reached.

Error Conditions

The `feof` function does not return any error condition.

Example

```
#include <stdio.h>
void print_char_from_file(FILE *fp)
{
   /* printf out each character from a file until EOF */
   while (!feof(fp))
      printf("%c", fgetc(fp));
   printf("\n");
}
```

See Also

clearerr

ferror

test for read or write errors

Synopsis

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

The `ferror` function returns a non-zero value if an error has previously occurred reading from or writing to `stream`. If no errors have occurred on `stream` the return value is zero.

Error Conditions

The `ferror` function does not return any error condition.

Example

```
#include <stdio.h>
void test_for_error(FILE *fp)
{
   if (ferror(fp))
      printf("Error with read/write to stream\n");
   else
      printf("read/write to stream OKAY\n");
}
```

See Also

clearerr, feof

fir

finite impulse response filter

Synopsis

```
#include <filter.h>
void fir (x,y,n,s)
const float x[];      /* Input sample vector x             */
float y[];            /* Output sample vector y            */
int n;                /* Number of input samples           */
fir_state *s;         /* Pointer to filter state structure */
```

The FIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
   float *h;          /* Filter coefficients            */
   float *d;          /* Start of delay line            */
   float *p;          /* Read/Write pointer             */
   int k;             /* Number of coefficients         */
   int l;             /* Interpolation/decimation index */
} fir_state;
```

Description

The `fir` function implements a finite impulse response (FIR) filter. The function generates the filtered response of the input data `x` and stores the result in the output vector `y`. The number of input samples and the length of the output vector is specified by the argument `n`.

The function maintains the filter state in the structured variable `s`, which must be declared and initialized before calling the function. The macro `fir_init`, in the `filter.h` header file, is available to initialize the structure and is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs. index)  \
   (state).h = (coeffs);    \
   (state).d = (delay);     \
   (state).p = (delay);     \
   (state).k = (ncoeffs);   \
   (state).l = (index)
```

The characteristics of the filter (passband, stopband, and so on) are dependent upon the number of filter coefficients and their values. A pointer to the coefficients should be stored in `s->h`, and `s->k` should be set to the number of coefficients.

Each filter should have its own delay line which is a vector of type `float` and whose length is equal to the number of coefficients. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `s->d` should be set to the start of the delay line, and the function uses `s->p` to keep track of its current position within the vector.

The structure member `s->l` is not used by `fir`. This field is normally set to an interpolation/decimation index before calling either the `fir_interp` or `fir_decima` functions.

The `fir` function assumes that the input data and the vector containing the filter coefficients are aligned on a quad-word boundary. For optimal performance, the delay line and the output buffer should not be in the same memory block as the filter coefficients.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} h(i) * x(k-i) \text{ for } k = 0,1,...n-1$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

fir_decima

FIR decimation filter

Synopsis

```
#include <filter.h>
void fir_decima (x,y,ng,s)
const float x[];        /* Input sample vector x             */
float y[];              /* Output sample vector y            */
int ng;                 /* Number of samples to generate     */
fir_state *s;           /* Pointer to filter state structure */
```

The FIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
   float *h;            /* Filter coefficients           */
   float *d;            /* Start of delay line           */
   float *p;            /* Read/Write pointer            */
   int k;               /* Number of coefficients        */
   int l;               /* Interpolation/decimation index */
} fir_state;
```

Description

The `fir_decima` function performs an FIR-based decimation filter. It generates the filtered decimated response of the input data `x` and stores the result in the output vector `y`. The size of the output vector is specified by the argument `ng`, and the number of input samples should be `ng*l`, where `l` is the decimation index.

The function maintains the filter state in the structured variable `s`, which must be declared and initialized before calling the function. The macro `fir_init`, in the `filter.h` header file, is available to initialize the structure and is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs, index) \
   (state).h = (coeffs);  \
   (state).d = (delay);   \
   (state).p = (delay);   \
   (state).k = (ncoeffs); \
   (state).l = (index)
```

The characteristics of the filter are dependent upon the number of filter coefficients and their values, and on the decimation index supplied by the calling program. A pointer to the coefficients should be stored in `s->h`, and `s->k` should be set to the number of coefficients. The decimation index is supplied to the function in `s->l`.

Each filter should have its own delay line which is a vector of type `float` and whose length is equal to the number of coefficients. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `s->d` should be set to the start of the delay line, and the function uses `s->p` to keep track of its current position within the vector.

The `fir_decima` function requires that the filter coefficients are aligned on a quad-word boundary, and that the coefficients are stored in reverse order, thus `s->h[0]` contains the last coefficient.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} x(k * l - i) * h(i)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

fir_interp

FIR interpolation filter

Synopsis

```
#include <filter.h>
void fir_interp (x,y,n,s)
const float x[];          /* Input sample vector x             */
float y[];                /* Output sample vector y            */
int n;                    /* Number of input samples           */
fir_state *s;             /* Pointer to filter state structure */
```

The FIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
   float *h;              /* Filter coefficients           */
   float *d;              /* Start of delay line           */
   float *p;              /* Read/Write pointer            */
   int k;                 /* Coefficients per polyphase    */
   int l;                 /* Interpolation factor          */
} fir_state;
```

Description

The `fir_interp` function performs a polyphase interpolation filter. It generates the interpolated filtered response of the input data `x` and stores the result in the output vector `y`. The number of input samples is specified by the argument `n`, and the size of the output vector should be `n*l`, where `l` is the interpolation index.

The filter characteristics are dependent upon the number of filter coefficients and their values, and on the interpolation factor supplied by the calling program. Each set of polyphase filter coefficients should be stored continually in reverse order. Therefore, if the filter coefficients have been

generated in normal order by a filter design tool, they have to be re-ordered before they are passed to the filter, as illustrated by the following formula:

```
filter_coeffs [(nc * nphases) - np]
```

where:

```
nc = {1, 2, ..., ncoeffs}
np = {1, 2, ..., nphases}
```

`ncoeffs` represents the number of coefficients per polyphase filter
`nphases` represents the number of polyphase filters and is set to the interpolation factor

For example, if the number of polyphase filters is 4 and the total number of coefficients is 12, then the coefficients should be ordered as shown below:

```
filter_coeffs[nphases-1],
filter_coeffs[(2*nphases)-1],
filter_coeffs[(3*nphases)-1],
...
filter_coeffs[nphases-4],
filter_coeffs[(2*nphases)-4],
filter_coeffs[(3*nphases)-4].
```

The `fir_interp` function assumes that the filter coefficients are aligned on a quad-word boundary.

A pointer to the coefficients is passed into the `fir_interp` function via the argument `s`, which is a structured variable that represents the filter state. This structured variable must be declared and initialized before calling the function. The `filter.h` header file contains the macro `fir_init` that can be used to initialize the variable and is defined as:

```
#define fir_init(state, coeffs, delay, ncoeffs, index) \
   (state).h = (coeffs);   \
   (state).d = (delay);    \
   (state).p = (delay);    \
   (state).k = (ncoeffs);  \
   (state).l = (index)
```

The interpolation factor is supplied to the function in `s->l`. A pointer to the coefficients should be stored in `s->h`, and `s->k` should be set to the number of coefficients per polyphase filter.

Each filter should have its own delay line which is a vector of type `float` and whose length is equal to the number of coefficients. The vector should be cleared to zero before calling the function for the first time and should not otherwise be modified by the user program. The structure member `s->d` should be set to the start of the delay line, and the function uses `s->p`, the read/write pointer, to keep track of its current position within the vector.

The output returned by the function is multiplied by the number of polyphase filters. Therefore, each element of the output vector has to be scaled accordingly. This is demonstrated in the **Example** below.

Algorithm

$$y_m(k) = \sum_{i=0}^{p/l-1} x(k-i) * h(i*l+m)$$

where m={0,1,2,...,l}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

Example

```
#include <filter.h>

#define INTERP_FACTOR 2
#define NCOEFFS       24
#define NSAMPLES      128
#define NPOLY         INTERP_FACTOR

/* Coefficients in normal order */

float filter_coeffs[NCOEFFS];

/* Coefficients in implementation order */

#pragma align 4
float coeffs[NCOEFFS];

/* Input, Output, Delay Line, and Filter State */

float input[NSAMPLES], output[INTERP_FACTOR*NSAMPLES];
float delay[NCOEFFS];

fir_state state;

/* Utility Variables */

int scale;
int i,nc,np;

/* Transform the normal order coefficients from a filter design
   tool into coefficients for the fir_interp function */

for (np = 1, i = 0; np <= NPOLY; np++)
    for (nc = 1; nc <= (NCOEFFS/NPOLY); nc++)
        coeffs[i++] = filter_coeffs[(nc * NPOLY) - np];

/* Initialize the delay line */

for (i = 0; i < NCOEFFS; i++)
```

```
    delay[i] = 0;

/* Initialize the filter state */

fir_init (state, coeffs, delay, (NCOEFFS/NPOLY), INTERP_FACTOR);

/* Call the fir_interp function */

fir_interp (input, output, NSAMPLES, &state);

/* Adjust output */

scale = NPOLY;
for (i = 0; i < (INTERP_FACTOR*NSAMPLES); i++)
    output[i] = output[i] / scale;
```

fflush

flush a stream

Synopsis

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

The `fflush` function causes any unwritten data for `stream` to be written to the file. If `stream` is a `NULL` pointer, `fflush` performs this flushing action on all streams.

Upon successful completion, the `fflush` function returns zero.

Error Conditions

If `fflush` is unsuccessful, the `EOF` value is returned.

Example

```
#include <stdio.h>
void flush_all_streams(void)
{
   fflush(NULL);
}
```

See Also

fclose

fgetc

get a character from a stream

Synopsis

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

The `fgetc` function obtains the next character from the input stream pointed to by `stream`, converts it from an `unsigned char` to an `int`, and advances the file position indicator for the stream.

Upon successful completion the `fgetc` function will return the next byte from the input stream pointed to by `stream`.

Error Conditions

If the `fgetc` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>
char use_fgetc(FILE *fp)
{
   char ch;
   if ((ch = fgetc(fp)) == EOF) {
      printf("Read End-of-file\n")
      return 0;
   } else {
      return ch;
   }
}
```

See Also

getc

fgetpos

record current position in a stream

Synopsis

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

The `fgetpos` function stores the current value of the file position indicator for the stream pointed to by `stream` in the file position type object pointed to by `pos`. The information generated by `fgetpos` in `pos` can be used with the `fsetpos` function to return the file to this position.

Upon successful completion, the `fgetpos` function will return a value of zero.

Error Conditions

If `fgetpos` is unsuccessful, the function will return a non-zero value.

Example

```
#include <stdio.h>
void aroutine(FILE *fp, char *buffer)
{
   fpos_t pos;
   /* get the current file position */
   if (fgetpos(fp, &pos)!= 0) {
      printf("fgetpos failed\n");
      return;
   }
   /* write the buffer to the file */
   (void) fprintf(fp, "%s\n", buffer);
   /* reset the file position to the value before the write */
   if (fsetpos(fp, &pos) != 0) {
      printf("fsetpos failed\n");
   }
}
```

See Also

fsetpos, ftell, fseek, rewind

fgets

get a string from a stream

Synopsis

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Description

The `fgets` function reads characters from `stream` into the array pointed to by `s`. The function will read a maximum of one less character than the value specified by `n`, although the get will also end if either a `NEWLINE` character or the end-of-file marker are read. The array `s` will have a `NUL` character written at the end of the string that has been read.

Upon successful completion, the `fgets` function will return `s`.

Error Conditions

If `fgets` is unsuccessful, the function will return a `NULL` pointer.

Example

```
#include <stdio.h>
char buffer[20];
void read_into_buffer(FILE *fp)
{
   char *str;

   str = fgets(buffer, sizeof(buffer), fp);
   if (str == NULL) {
      printf("Either read failed or EOF encountered\n");
   } else {
      printf("filled buffer with %s\n", str);
   }
}
```

See Also

fgetc, getc, gets

floor

floor

Synopsis

```
#include <math.h>
float floorf (float x)
double floor (double x)
long double floord (long double x)
```

Description

The `floor` functions return the largest integral value that is not greater than their argument.

Algorithm

return = *largest int* < x

Domain

x = [-3.4×10^{38} ... 3.4×10^{38}] for `floorf()`

x = [-1.7×10^{308} ... 1.7×10^{308}] for `floord()`

fmax

maximum

Synopsis

```
#include <math.h>

float fmaxf (float parm1, float parm2);
double fmax (double parm1, double parm2);
long double fmaxd (long double parm1, long double parm2);
```

Description

The `fmax` functions return the larger of their two arguments.

The `fmaxf` function is a built-in function which is implemented with a `MAX` instruction. The `fmax` function is compiled as a built-in function if `double` is the same size as `float`.

Algorithm

```
if ( parm1 > parm2 )
   return (parm1)
else
   return (parm2)
```

Domain

Full range.

fmin

minimum

Synopsis

```
#include <math.h>

float fminf (float parm1, float parm2);
double fmin (double parm1, double parm2);
long double fmind (long double parm1, long double parm2);
```

Description

The `fmin` functions return the smaller of their two arguments.

The `fminf` function is a built-in function which is implemented with a `MIN` instruction. The `fmin` function is compiled as a built-in function if `double` is the same size as `float`.

Algorithm

```
if ( parm1 < parm2 )
   return (parm1)
else
   return (parm2)
```

Domain

Full range.

fmod

floating-point modulus

Synopsis

```
#include <math.h>

float fmodf (float x, float y)
double fmod (double x, double y)
long double fmodd (long double x, long double y)
```

Description

The `fmod` function computes the floating-point remainder that results from dividing the first argument into the second argument. This value is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, `fmod` returns a zero.

Algorithm

$$return = \left(\left| \frac{x}{y} \right| - floor \left| \frac{x}{y} \right| \right) \cdot sign(x)$$

Domain

x = $[-3.4 \times 10^{38} \ldots 3.4 \times 10^{38}]$ for `fmodf()`

x = $[-1.7 \times 10^{308} \ldots 1.7 \times 10^{308}]$ for `fmodd()`

NOT y = 0

fopen

open a file

Synopsis

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Description

The `fopen` function initializes the data structures that are required for reading or writing to a file. The file's name is identified by `filename`, with the access type required specified by the string `mode`.

Valid selections for `mode` are specified below. If any other mode specification is selected then the behavior is undefined.

Table 3-18. Valid Selections for mode

mode	Selection
`r`	Open text file for reading. This operation fails if the file has not previously been created.
`w`	Open text file for writing. If the filename already exists then it will be truncated to zero length with the write starting at the beginning of the file. If the file does not already exist then it is created.
`a`	Open a text file for appending data. All data will be written to the end of the file specified.
`r+`	As `r` with the exception that the file can also be written to.
`w+`	As `w` with the exception that the file can also be read from.
`a+`	As `a` with the exception that the file can also be read from any position within the file. Data is only written to the end of the file.
`rb`	As `r` with the exception that the file is opened in binary mode.
`wb`	As `w` with the exception that the file is opened in binary mode.
`ab`	As `a` with the exception that the file is opened in binary mode.

Table 3-18. Valid Selections for mode (Cont'd)

mode	Selection
r+b/rb+	Open file in binary mode for both reading and writing.
w+b/wb+	Create or truncate to zero length a file for both reading and writing.
a+b/ab+	As a+ with the exception that the file is opened in binary mode.

If the call to the fopen function is successful, a pointer to the object controlling the stream is returned.

Error Conditions

If the fopen function is unsuccessful, a NULL pointer is returned.

Example

```
#include <stdio.h>

FILE *open_output_file(void)
{
   /* Open file for writing as binary */
   FILE *handle = fopen("output.dat", "wb");
   return handle;
}
```

See Also

fclose, fflush, freopen

fprintf

print formatted output

Synopsis

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, /*args*/ ...);
```

Description

The `fprintf` function places output on the named output `stream`. The string pointed to by `format` specifies how the arguments are converted for output.

The format string can contain zero or more conversion specifications, each beginning with the `%` character. The conversion specification itself follows the `%` character and consists of one or more of the following sequences:

- Flag - optional characters that modifies the meaning of the conversion.
- Width - optional numeric value (or *) that specifies the minimum field width.
- Precision - optional numeric value that gives the minimum number of digits to appear.
- Length - optional modifier that specifies the size of the argument.
- Type - character that specifies the type of conversion to be applied.

The flag characters can be in any order and are optional. The valid flags are described in the following table:

The minimum field width is an optional value, specified as a decimal number. If a field width is specified then the converted value is padded with spaces to the specified width if the result contains fewer characters than width. If the width field value begins with `0` then zeros are used to

Table 3-19. Valid Flags for `fprintf`

Flag	Field
-	Left justify the result within the field. The result is right-justified by default.
+	Always begin a signed conversion with a plus or minus sign. By default only negative values will start with a sign.
space	Prefix a space to the result if the first character is not a sign and the + flag has not also been specified.
#	The result is converted to an alternative form depending on the type of conversion: o : If the value is not zero it is preceded with 0. x : If the value is not zero it is preceded with 0x. X : If the value is not zero it is preceded with 0X. a A e E f F: Always generate a decimal point. g G : as E except trailing zeros are not removed.

pad the field rather than spaces. A * in the width indicates that the width is specified by an integer value preceding the argument that has to be formatted.

The optional precision value always begins with a period (.) and is followed either by an asterisk (*) or by a decimal integer. An asterisk (*) indicates that the precision is specified by an integer argument preceding the argument to be formatted. If only a period is specified, a precision of zero will be assumed. The precision value has differing effects depending on the conversion specifier being used:

- For A, a specifies the number of digits after the decimal point. If the precision is zero and the # flag is not specified no decimal point will be generated.
- For d,i,o,u,x,X specifies the minimum number of digits to appear, defaulting to 1. No characters will be generated when the value to be output and also the precision is zero.

- For f,F,E,e specifies the number of digits after the decimal point character, the default being 6. If the # specifier is present with a zero precision then no decimal point will be generated.
- For g, G specifies the maximum number of significant digits.
- For s specifies the maximum number of characters to be written.

The length modifier can optionally be used to specify the size of the argument. The length modifiers should only precede one of the d, i, o, u, x, X or n conversion specifiers unless other conversion specifiers are detailed.

Table 3-20. Length Modifier Actions for fprintf

Length	Action
h	The argument should be interpreted as a short int.
l	The argument should be interpreted as a long int.
ll	The argument should be interpreted as a long long int.
L	The argument should be interpreted as a long double argument. This length modifier should precede one of the a, A, e, E, f, F, g, or G conversion specifiers. Note that this length modifier is only valid if -double-size-64 is selected. If -double-size-32 is selected no conversion will occur, with the corresponding argument being consumed.

A definition of the valid conversion specifiers that define the type of conversion to be applied can be found in the following table:

Table 3-21. Conversion Specifiers Characters for fprintf

Specifier	Conversion
a, A	floating-point number
c	character
d, i	signed decimal integer
e, E	scientific notation (mantissa/exponent)

Table 3-21. Conversion Specifiers Characters for fprintf (Cont'd)

Specifier	Conversion
f, F	decimal floating-point
g, G	convert as e, E or f, F
n	pointer to signed integer to which the number of characters written so far will be stored with no other output
o	unsigned octal
p	pointer to void
s	string of characters
u	unsigned integer
x, X	unsigned hexadecimal notation
%	print a % character with no argument conversion

The a|A conversion specifier converts to a floating-point number with the notational style [-]0xh.hhhh±d where there is one hexadecimal digit before the period. The a|A conversion specifiers always contain a minimum of one digit for the exponent.

The e|E conversion specifier converts to a floating-point number notational style [-]d.ddde±dd. The exponent always contains at least two digits. The case of the e preceding the exponent will match that of the conversion specifier.

The f|F conversion specifier converts to decimal notation [-]d.ddd±ddd.

The g|G conversion specifier converts as e|E or f|F specifiers depending on the value being converted. If the value being converted is less than -4 or greater than or equal to the precision then e|E conversions will be used, otherwise f|F conversions will be used.

For all of the a, A, e, E, f, F, g and G specifiers, an argument that represents infinity is displayed as Inf. For all of the a, A, e, E, f, F, g and G specifiers, an argument that represents a NaN result is displayed as NaN.

The `fprintf` function returns the number of characters printed.

Error Conditions

If the `fprintf` function is unsuccessful, a negative value is returned.

Example

```
#include <stdio.h>

void fprintf_example(void)
{
   char *str = “hello world”;
   /* Output to stdout is "   +1  +1." */
   fprintf(stdout, "%+5.0f%+#5.0f\n", 1.234, 1.234);

   /* Output to stdout is "1.234 1.234000 1.23400000" */
   fprintf(stdout, "%.3f %f %.8f\n", 1.234, 1.234, 1.234);

   /* Output to stdout is "justified:
                           left:5    right:    5" */
   fprintf(stdout, "justified:\nleft:%-5dright:%5i\n", 5, 5);

   /* Output to stdout is
      “90% of test programs print hello world" */
   fprintf(stdout, "90%% of test programs print %s\n", str);

   /* Output to stdout is "0.0001 1e-05 100000 1E+06" */
   fprintf(stdout, "%g %g %G %G\n", 0.0001, 0.00001, 1e5, 1e6);
}
```

See Also

printf, snprintf, vfprintf, vprintf, vsnprintf, vsprintf

fputc

put a character on a stream

Synopsis

```
#include <stdio.h>
int fputc(int ch, FILE *stream);
```

Description

The `fputc` function writes the argument `ch` to the output stream pointed to by `stream` and advances the file position indicator. The argument `ch` is converted to an `unsigned char` before it is written.

If the `fputc` function is successful, it will return the value that was written to the stream.

Error Conditions

If the `fputc` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

void fputc_example(FILE* fp)
{
   /* put the character 'i' to the stream pointed to by fp */
   int res = fputc('i', fp);
   if (res != 'i')
      printf("fputc failed\n");
}
```

See Also

putc

fputs

put a string on a stream

Synopsis

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

Description

The `fputs` function writes the string pointed to by `string` to the output stream pointed to by `stream`. The `NULL` terminating character of the string will not be written to `stream`.

If the call to `fputs` is successful, the function will return a non-negative value.

Error Conditions

The `fputs` function will return `EOF` if a write error occurred.

Example

```
#include <stdio.h>

void fputs_example(FILE* fp)
{
   /* put the string "example" to the stream pointed to by fp */
   char *example = "example";
   int res = fputs(example, fp);
   if (res == EOF)
      printf("fputs failed\n");
}
```

See Also

puts

fread

buffered input

Synopsis

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Description

The `fread` function reads into an array pointed to by `ptr` up to a maximum of `n` items of data from `stream`, where an item of data is a sequence of bytes of length `size`. It stops reading bytes if an `EOF` or error condition is encountered while reading from `stream`, or if `n` items have been read. It advances the data pointer in `stream` by the number of bytes read. It does not change the contents of `stream`.

The `fread` function returns the number of items read. This may be less than `n` if there is insufficient data on the external device to satisfy the read request. If `size` or `n` is zero, then `fread` will return zero and does not affect the state of `stream`.

When the stream has been opened as a binary stream, the Analog Devices I/O library may choose to bypass the I/O buffer and transmit data from an external device directly into the program, particularly when the buffer size (as defined by the macro `BUFSIZ` in the `stdio.h` header file, or controlled by the function `setvbuf`) is smaller than the number of characters to be transferred.

Error Conditions

If an error occurs, `fread` will return zero and set the error indicator for `stream`.

Example

```
#include <stdio.h>
int buffer[100];

int fill_buffer(FILE *fp)
{
   int read_items;
   /* Read from file pointer fp into array buffer  */
   read_items = fread(&buffer, sizeof(int), 100, fp);
   if (read_items < 100) {
       if (ferror(fp))
           printf("fill_buffer failed with an I/O error\n");
        else if (feof(fp))
           printf("fill_buffer failed with EOF\n");
       else
           printf("fill_buffer only read %d items\n",read_items);
   }
   return read_items;
}
```

See Also

ferror, fgetc, fgets, fscanf

freopen

open a file using an existing file descriptor

Synopsis

```
#include <stdio.h>
FILE *freopen(const char *fname, const char *mode, FILE *stream);
```

Description

The `freopen` function opens the file specified by `fname` and associates it with the stream pointed to by `stream`. The mode argument has the same effect as described in `fopen` (See "fopen" on page 3-182 for more information on the mode argument).

Before opening the new file, the `freopen` function will first attempt to flush the stream and close any file descriptor associated with `stream`. Failure to flush or close the file successfully is ignored. Both the error and `EOF` indicators for `stream` are cleared.

The original stream will always be closed regardless of whether the opening of the new file is successful or not.

Upon successful completion, the `freopen` function returns the value of `stream`.

Error Conditions

If `freopen` is unsuccessful, a `NULL` pointer is returned.

Example

```
#include <stdio.h>

void freopen_example(FILE* fp)
{
   FILE *result;
   char *newname = "newname";
```

```
    /* reopen existing file pointer for reading file "newname" */
    result = freopen(newname, "r", fp);
    if (result == fp)
       printf("%s reopened for reading\n", newname);
    else
       printf("freopen not successful\n");
}
```

See Also

fclose, fopen

frexp

separate fraction and exponent

Synopsis

```
#include <math.h>
float frexpf (float x, int *n)
double frexp (double x, int *n)
long double frexpd (long double x, int *n)
```

Description

The `frexp` function separates a floating-point input into a normalized fraction and a (base 2) exponent. The function returns the first argument as a fraction in the interval [½, 1), and stores a power of 2 in the integer pointed to by the second argument. If the input is zero, then the fraction and exponent are both set to zero.

Algorithm

`return = f;` `n` passed through 2nd parameter pointer. $x = f * 2^n$

Domain

x = [-3.4×10^{38} ... 3.4×10^{38}] for `frexpf()`

x = [-1.7×10^{308} ... 1.7×10^{308}] for `frexpd()`

fscanf

read formatted input

Synopsis

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, /* args */...);
```

Description

The `fscanf` function reads from the input file `stream`, interprets the inputs according to `format` and stores the results of the conversions in its arguments. The `format` is a string containing the control format for the input with the following arguments being pointers to the locations where the converted results are to be written to.

The string pointed to by `format` specifies how the input is to be parsed and, possibly, converted. It may consist of whitespace characters, ordinary characters (apart from the `%` character), and conversion specifications. A sequence of whitespace characters causes `fscanf` to continue to parse the input until either there is no more input or until it find a non-whitespace character. If the format specification contains a sequence of ordinary characters, `fscanf` will continue to read the next characters in the input stream until the input data does not match the sequence of characters in the format. At this point `fscanf` will fail, and the differing and subsequent characters in the input stream will not be read.

The `%` character in the format string introduces a conversion specification. A conversion specification has the following form:

```
% [*] [width] [length] type
```

A conversion specification always starts with the `%` character. It may optionally be followed by an asterisk (`*`) character which indicates that the result of the conversion is not to be saved. In this context the asterisk character is known as the assignment-suppressing character. The optional

token `width` represents a non-zero decimal number and specifies the maximum field width. `fscanf` will not read any more than `width` characters while performing the conversion specified by `type`. The `length` token can be used to define a length modifier.

The `length` modifier can be used to specify the size of the argument. The length modifiers should only precede one of the `d`, `e`, `f`, `g`, `i`, `o`, `u`, `x` or `n` conversion specifiers unless other conversion specifiers are detailed.

Table 3-22. Length Modifier for `fscanf`

Length	Action
`h`	The argument should be interpreted as a `short` int.
`l`	The argument should be interpreted as a `long` int.
`ll`	The argument should be interpreted as a `long long` int.
`L`	The argument should be interpreted as a `long double` argument. This length modifier should precede one of the `a, A, e, E, f, F, g,` or `G` conversion specifiers.

A definition of the valid conversion specifier characters that specify the type of conversion to be applied can be found in the following table:

Table 3-23. Conversion Specifier Characters for `fscanf`

Specifier	Conversion
`a A e E f F g G`	floating point, optionally preceded by a sign and optionally followed by an e or E character
`c`	single character, including whitespace
`d`	signed decimal integer with optional sign
`i`	signed integer with optional sign
`n`	no input is consumed. The number of characters read so far will be written to the corresponding argument. This specifier does not affect the function result returned by `fscanf`
`o`	unsigned octal

Table 3-23. Conversion Specifier Characters for `fscanf`

Specifier	Conversion
`p`	pointer to void
s	string of characters up to a whitespace character
`u`	unsigned decimal integer
`x X`	hexadecimal integer with optional sign
`[`	a non-empty sequence of characters referred to as the scanset
`%`	a single `%` character with no conversion or assignment

The `[` conversion specifier should be followed by a sequence of characters, referred to as the `scanset`, with a terminating `]` character. It will take the form `[scanset]`. The conversion specifier copies into an array that is the corresponding argument until a character that does not match any of the scanset is read. If the scanset begins with a `^` character, the scanning will match against characters not defined in the scanset. If the scanset is to include the `]` character, this character must immediately follow the `[` character or the `^` character (if specified).

Each input item is converted to a type appropriate to the conversion character, as specified in the table above. The result of the conversion is placed into the object pointed to by the next argument that has not already been the recipient of a conversion. If the suppression character has been specified, no data shall be placed into the object with the next conversion, using the object to store its result.

The `fscanf` function returns the number of items successfully read.

Error Conditions

If the `fscanf` function is unsuccessful, before any conversion then `EOF` is returned.

Example

```
#include <stdio.h>

void fscanf_example(FILE *fp)
{
   short int day, month, year;
   float f1, f2, f3;
   char string[20];

   /* Scan a date with any separator, eg, 1-1-2006 or 1/1/2006 */
   fscanf (fp, "%hd%*c%hd%*c%hd", &day, &month, &year);

   /* Scan float values separated by “abc”, for example
      1.234e+6abc1.234abc234.56abc                        */
   fscanf (fp, "%fabc%gabc%eabc", &f1, &f2, &f3);

   /* For input “alphabet”, string will contain “a” */
   fscanf (fp, "%[aeiou]", string);

   /* For input “drying”, string will contain “dry” */
   fscanf (fp, "%[^aeiou]", string);
}
```

See Also

scanf, sscanf

fseek

reposition a file position indicator in a stream

Synopsis

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Description

The `fseek` function sets the file position indicator for the stream pointed to by `stream`. The position within the file is calculated by adding the offset to a position dependent on the value of `whence`. The valid values and effects for `whence` are as follows:

Table 3-24. Valid Values and Effects for `whence` Parameter

whence	Effect
SEEK_SET	Set the position indicator to be equal to `offset` bytes from the beginning of `stream`.
SEEK_CUR	Set the new position indicator to current position indicator for `stream` plus `offset`.
SEEK_END	Set the position indicator to `EOF` plus `offset`.

Using `fseek` to position a text stream is valid only if either `offset` is zero, or if `whence` is `SEEK_SET` and `offset` is a value that was previously returned by `ftell`.

Positioning within a file that has been opened as a text stream is supported only by the libraries that Analog Devices supply if the lines within the file are terminated by the character sequence `\r\n`.

A successful call to `fseek` will clear the `EOF` indicator for `stream` and undoes any effects of `ungetc` on `stream`. If the stream has been opened as a update stream, then the next I/O operation may be either a read request or a write request.

Error Conditions

If the `fseek` function is unsuccessful, a non-zero value is returned.

Example

```
#include <stdio.h>

long fseek_and_ftell(FILE *fp)
{
   long offset;
   /* seek to 20 bytes offset from given file pointer */
   if (fseek(fp, 20, SEEK_SET) != 0) {
      printf("fseek failed\n");
      return -1;
   }
   /* Now use ftell to get the offset value back */
   offset = ftell(fp);
   if (offset == -1)
       printf("ftell failed\n");
   if (offset == 20)
       printf("ftell and fseek work\n");
   return offset;
}
```

See Also

fflush, ftell, ungetc

fsetpos

reposition a file pointer in a stream

Synopsis

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

The `fsetpos` function sets the file position indicator for `stream`, using the value of the object pointed to by `pos`. The value pointed to by `pos` must be a value obtained from an earlier call to `fgetpos` on the same stream.

Positioning within a file that has been opened as a text stream is supported only by the libraries that Analog Devices supply if the lines within the file are terminated by the character sequence `\r\n`.

A successful call to `fsetpos` function clears the `EOF` indicator for `stream` and undoes any effects of `ungetc` on the same stream.

The `fsetpos` function returns zero if it is successful.

Error Conditions

If the `fsetpos` function is unsuccessful, the function returns a non-zero value.

Example

Refer to "fgetpos" on page 3-174.

See Also

fgetpos, fseek, ftell, rewind, ungetc

ftell

obtain current file position

Synopsis

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

The `ftell` function obtains the current position for a file identified by `stream`.

If `stream` is a binary stream, the value is the number of characters from the beginning of the file. If `stream` is a text stream, the information in the position indicator is unspecified information that is usable by `fseek` for determining the file position indicator at the time of the `ftell` call.

Positioning within a file that has been opened as a text stream is supported only by the libraries that Analog Devices supply if the lines within the file are terminated by the character sequence `\r\n`.

If successful, the `ftell` function returns the current value of the file position indicator on the stream.

Error Conditions

If the `ftell` function is unsuccessful, a value of -1 is returned.

Example

See `fseek` on page 3-201 for an example.

See Also

fgetpos, fseek

fwrite

buffered binary output

Synopsis

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t n,
              FILE *stream);
```

Description

The `fwrite` function writes to the output `stream` up to `n` items of data from the array pointed by `ptr`. An item of data is defined as a sequence of characters of size `size`. The write will complete once `n` items of data have been written to the stream. The file position indicator for `stream` is advanced by the number of characters successfully written.

When the stream has been opened as a binary stream, the Analog Devices I/O library may choose to bypass the I/O buffer and transmit data from the program directly to the external device, particularly when the buffer size (as defined by the macro `BUFSIZ` in the `stdio.h` header file, or controlled by the function `setvbuf`) is smaller than the number of characters to be transferred.

If successful then the `fwrite` function will return the number of items written.

Error Conditions

If the `fwrite` function is unsuccessful, it will return the number of elements successfully written which will be less than `n`.

Example

```
#include <stdio.h>
#include <stdlib.h>

char* message="some text";

void write_text_to_file(void)
{
   /* Open "file.txt" for writing */
   FILE* fp = fopen("file.txt", "w");
   int res, message_len = strlen(message);
   if (!fp) {
      printf("fopen was not successful\n");
      return;
   }
   res = fwrite(message, sizeof(char), message_len, fp);
   if (res != message_len)
      printf("fwrite was not successful\n");
}
```

See Also

fread

gen_bartlett

generate Bartlett window

Synopsis

```
#include <window.h>
void gen_bartlett (w,a,N)
float w[];     /* Window vector                                  */
int a;         /* Address stride in samples for window vector */
int N;         /* Length of window vector                        */
```

Description

This function generates a vector containing the Bartlett window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

The Bartlett window is similar to the Triangle window (described on page 3-207) but has the following different properties:

- The Bartlett window always returns a window with two zeros on either end of the sequence, so that for odd `n`, the center section of an `N+2` Bartlett window equals an `N` Triangle window.
- For even `n`, the Bartlett window is the convolution of two rectangular sequences. There is no standard definition for the Triangle window for even `n`; the slopes of the Triangle window are slightly steeper than those of the Bartlett window.

Algorithm

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

gen_blackman

generate Blackman window

Synopsis

```
#include <window.h>
void gen_blackman (w,a,N)
float w[];     /* Window vector                                    */
int a;         /* Address stride in samples for window vector */
int N;         /* Length of window vector                          */
```

Description

This function generates a vector containing the Blackman window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

$$w[n] = 0.42 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) + 0.08\cos\left(\frac{4\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

gen_gaussian

generate Gaussian window

Synopsis

```
#include <window.h>
void gen_gaussian (w,alpha,a,N)
float w[];     /* Window vector                                    */
float alpha;   /* Gaussian alpha parameter                         */
int a;         /* Address stride in samples for window vector      */
int N;         /* Length of window vector                          */
```

Description

This function generates a vector containing the Gaussian window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

$$w(n) = \exp\left[-\frac{1}{2}\left(\alpha\,\frac{n - N/2 - 1/2}{N/2}\right)^2\right]$$

where n = {0, 1, 2, ..., N-1} and α is an input parameter

Domain

$a > 0$; $N > 0$; $\alpha > 0.0$

gen_hamming

generate Hamming window

Synopsis

```
#include <window.h>
void gen_hamming (w,a,N)
float w[];    /* Window vector                                    */
int a;        /* Address stride in samples for window vector */
int N;        /* Length of window vector                          */
```

Description

This function generates a vector containing the Hamming window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

$$w[n] = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

gen_hanning

generate Hanning window

Synopsis

```
#include <window.h>
void gen_hanning (w,a,N)
float w[];     /* Window vector                                    */
int a;         /* Address stride in samples for window vector */
int N;         /* Length of window vector                          */
```

Description

This function generates a vector containing the Hanning window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Cosine window.

Algorithm

$$w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

gen_harris

generate Harris window

Synopsis

```
#include <window.h>
void gen_harris (w,a,N)
float w[];     /* Window vector                                    */
int a;         /* Address stride in samples for window vector */
int N;         /* Length of window vector                          */
```

Description

This function generates a vector containing the Harris window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Blackman-Harris window.

Algorithm

$$w[n] = 0.35875 - 0.48829 * \cos\left(\frac{2\pi n}{N-1}\right) + 0.14128 * \cos\left(\frac{4\pi n}{N-1}\right) - 0.01168 * \cos\left(\frac{6\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

gen_kaiser

generate Kaiser window

Synopsis

```
#include <window.h>
void gen_kaiser (w,beta,a,N)
float w[];     /* Window vector                                    */
float beta;    /* Kaiser beta parameter                            */
int a;         /* Address stride in samples for window vector */
int N;         /* Length of window vector                          */
```

Description

This function generates a vector containing the Kaiser window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. The β value is specified by parameter `beta`.

Algorithm

$$w[n] = \frac{I_0\left[\beta\left(1-\left[\frac{n-\alpha}{\alpha}\right]^2\right)^{1/2}\right]}{I_0(\beta)}$$

where n = {0, 1, 2, ..., N-1}, α= (N - 1) /2, and $I_0(\beta)$ represents the zeroth-order modified Bessel function of the first kind.

Domain

$\alpha > 0$; $N > 0$; $\beta > 0.0$

gen_rectangular

generate rectangular window

Synopsis

```
#include <window.h>
void gen_rectangular (w,a,N)
float w[];     /* Window vector                                    */
int a;         /* Address stride in samples for window vector */
int N;         /* Length of window vector                          */
```

Description

This function generates a vector containing the rectangular window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

$w[n] = 1$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

gen_triangle

generate triangle window

Synopsis

```
#include <window.h>
void gen_triangle (w,a,N)
float w[];      /* Window vector                                    */
int a;          /* Address stride in samples for window vector */
int N;          /* Length of window vector                          */
```

Description

This function generates a vector containing the Triangle window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Refer to the Bartlett window (described on page 3-207) regarding the relationship between it and the Triangle window.

Algorithm

For even `n`, the following equation applies:

$$w[n] = \begin{cases} \dfrac{2n+1}{N} & n < N/2 \\ \dfrac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where n = {0, 1, 2, ..., N-1}

For odd n, the following equation applies:

$$w[n] = \begin{cases} \dfrac{2n+2}{N+1} & n < N/2 \\ \dfrac{2N-2n}{N+1} & n > N/2 \end{cases}$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

gen_vonhann

generate von Hann window

Synopsis

```
#include <window.h>
void gen_vonhann (w,a,N)
float w[];      /* Window vector                                  */
int a;          /* Address stride in samples for window vector */
int N;          /* Length of window vector                        */
```

Description

This function is identical to the `gen_hanning` window (described on page 3-212).

Domain

a > 0; N > 0

getc

get a character from a stream

Synopsis

```
#include <stdio.h>
int getc(FILE *stream);
```

Description

The `getc` function is equivalent to `fgetc`. The `getc` function obtains the next character from the input stream pointed to by `stream`, converts it from an `unsigned char` to an `int`, and advances the file position indicator for the stream.

Upon successful completion, the `getc` function will return the next character from the input stream pointed to by `stream`.

Error Conditions

If the `getc` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

char use_getc(FILE *fp)
{
   char ch;
   if ((ch = getc(fp)) == EOF) {
      printf("Read End-of-file\n");
      return (char)-1;
   } else {
      return ch;
   }
}
```

See Also

fgetc

getchar

get a character from stdin

Synopsis

```
#include <stdio.h>
int getchar(void);
```

Description

The `getchar` function is functionally the same as calling the `getc` function with `stdin` as its argument. A call to `getchar` will return the next single character from the standard input stream. The `getchar` function also advances the standard input's current position indicator.

Error Conditions

If the `getchar` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

char use_getchar(void)
{
   char ch;
   if ((ch = getchar()) == EOF) {
      printf("getchar() failed\n");
      return (char)-1;
   } else {
      return ch;
   }
}
```

See Also

getc

gets

get a string from a stream

Synopsis

```
#include <stdio.h>
char *gets(char *s);
```

Description

The `gets` function reads characters from the standard input stream into the array pointed to by `s`. The read will terminate when a `NEWLINE` character is read, with the `NEWLINE` character being replaced by a null character in the array pointed to by `s`. The read will also halt if `EOF` is encountered.

The array pointed to by `s` must be of equal or greater length of the input line being read. If this is not the case, the behavior is undefined.

If `EOF` is encountered without any characters being read, a `NULL` pointer is returned.

Error Conditions

If the `gets` function is unsuccessful and a read error occurs, a `NULL` pointer is returned.

Example

```
#include <stdio.h>

void fill_buffer(char *buffer)
{
   if (gets(buffer) == NULL)
      printf("gets failed\n")
   else
      printf("gets read %s\n", buffer);
   }
}
```

See Also

fgetc, fgets, fread, fscanf

gmtime

convert calendar time into broken-down time as UTC

Synopsis

```
#include <time.h>
struct tm *gmtime (const time_t *t);
```

Description

The `gmtime` function converts a pointer to a calendar time into a broken-down time in terms of Coordinated Universal Time (UTC). A broken-down time is a structured variable, which is described in "time.h" on page 3-21.

The broken-down time is returned by `gmtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `gmtime`, or to `localtime`.

Error Conditions

The `gmtime` function does not return an error condition.

Example

```
#include <time.h>
#include <stdio.h>

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = gmtime(&cal_time);
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

localtime, mktime,time

histogramf

histogramf

Synopsis

```
#include <stats.h>

void histogramf (a,n,max,min,c,m)
const float a[];            /* Pointer to input vector a   */
int n;                      /* Number of input samples     */
float max;                  /* Maximum value of the bin    */
float min;                  /* Minimum value of the bin    */
int c[];                    /* Pointer to output vector c  */
int m;                      /* Number of bins              */
```

Description

The `histogramf` function computes a histogram of the input vector `a` that contains `n` samples, and stores the result in the output vector `c`.

The minimum and maximum value of any input sample is specified by `min` and `max`, respectively; these values are used by the function to calculate the size of each bin as `(max - min) / m` where `m` is the size of the output vector `c`.

Any input value that is outside the range `[ min , max )` exceeds the boundaries of the output vector and is discarded.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

The output vector is first zeroed by the function. Each input value is then adjusted by `min`, multiplied by `1/binsize`, and rounded to select the appropriate bin to increment.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

ifft

N point inverse FFT

Synopsis

```
#include <filter.h>
void ifft (in[], t[], out[], w[], wst, n)
const complex_float in[];      /* Input sequence           */
complex_float t[];             /* Temporary working buffer */
complex_float out[];           /* Output sequence          */
const complex_float w[];       /* Twiddle sequence         */
int wst;                       /* Twiddle factor stride    */
int n;                         /* Number of FFT points     */
```

Description

This function transforms the frequency domain complex input signal sequence to the time domain by using the accelerated version of the 'Discrete Fourier Transformation' known as an 'Inverse Fast Fourier Transform' or IFFT. It "decimates in frequency" by the best choice FFT algorithm, radix-4 or mixed-radix, depending on the input sequence length.

The size of the input array `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` represents the number of points in the FFT. If the input data can be overwritten, then the memory requirements may be reduced by specifying the input array as the output array. Run-time performance of the function is improved if the input and output arrays are allocated in a different memory block than the twiddle table, `w`.

The twiddle table is passed in the argument `w`, which must contain at least ¾`n` complex twiddle factors. The function `twidfft` may be used to initialize the array. If the twiddle table contains more factors than needed for a

particular call on `ifft`, then the stride factor has to be set appropriately; otherwise it should be 1. Refer to "twidfft" on page 3-336 for more information.

Algorithm

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) W_N^{-nk}$$

The implementation uses core FFT functions implemented as direct radix4, or direct mixed radix algorithm. To get the inverse effect, it first swaps the real and imaginary parts of the input, performs the direct radix4 or mixed radix transformation and finally swaps the real and imaginary parts of the output.

Domain

Input sequence length `n` must equal to either a power of two, or a power of four, and at least 16.

ifft2d

NxN point 2-D inverse input FFT

Synopsis

```
#include <filter.h>
void ifft2d (*in, *t, *out, w[], wst, n)
const complex_float *in;  /* Pointer to input matrix a[n][n]   */
complex_float *t;         /* Pointer to working buffer t[n][n] */
complex_float *out;       /* Pointer to matrix c[n][n]         */
const complex_float w[];  /* Twiddle sequence                  */
int wst;                  /* Twiddle factor stride             */
int n;                    /* Number of FFT points              */
```

Description

This function computes a two-dimensional inverse Fast Fourier Transform of the complex input matrix `a[n][n]` and stores the result in the complex matrix `c[n][n]`.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the "twiddle table" is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `ifft2d`, then the stride parameter needs to be set appropriately; otherwise, it should be one. For more information, see "twidfft" on page 3-336.

Algorithm

$$c(i,j) = \frac{1}{n^2} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a(k,l) * e^{2\pi j(i*k+j*l)/n}$$

where i={0,1,...,n -1}, j={0,1,2,...,n-1}

Domain

Input sequence length `n` must equal to either a power of two, or a power of four, and at least 16.

iir

infinite impulse response filter

Synopsis

```
#include <filter.h>
void iir (x,y,n,s)
const float x[];       /* Input sample vector x             */
float y[];             /* Output sample vector y            */
int n;                 /* Number of input samples           */
iir_state *s;          /* Pointer to filter state structure */
```

The IIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
   float *c;           /* Coefficients                      */
   float *d;           /* Start of delay line               */
   int k;              /* Number of biquad stages           */
} iir_state;
```

Description

The `iir` function implements a biquad, canonical form, infinite impulse response (IIR) filter. It generates the filtered response of the input data `x` and stores the result in the output vector `y`. The number of input samples and the length of the output vector is specified by the argument `n`.

The function maintains the filter state in the structured variable `s`, which must be declared and initialized before calling the function. The macro `iir_init`, in the `filter.h` header file, is available to initialize the structure and is defined as:

```
#define iir_init(state, coeffs, delay, stages)  \
   (state).c = (coeffs);   \
   (state).d = (delay);    \
   (state).k = (stages)
```

The characteristics of the filter are dependent upon the filter coefficients and the number of stages. Each stage has five coefficients which must be stored in the order `B2`, `B1`, `B0`, `A2`, `A1`. The value of `A0` is implied to be 1.0 and `A1` and `A2` should be scaled accordingly. A pointer to the coefficients should be stored in `s->c`, and `s->k` should be set to the number of stages.

Each filter should have its own delay line which is a vector of type `float` and whose length is equal to twice the number of stages. The vector should be initially cleared to zero and should not otherwise be modified by the user program. The structure member `s->d` should be set to the start of the delay line.

The `iir` function assumes that the delay line is quad-word aligned. There is no performance penalty if the input data, output vector, and filter state are allocated in the same memory block.

The `iir` function is implemented using a direct form II algorithm. When importing coefficients from a filter design tool that employs a transposed direct form II, the `A1` and `A2` coefficients have to be negated. For example, if a filter design tool returns `A= [1.0, 0.2, -0.9]`, then the A coefficients have to be modified to `A= [1.0, -0.2, 0.9]`.

Algorithm

$$H(z) = \frac{B_0 + B_1 z^{-1} + B_2 z^{-2}}{1 - A_1 z^{-1} - A_2 z^{-2}}$$

where

$$D_m = A_2 * D_{m-2} + A_1 * D_{m-1} + x_m$$
$$Y_m = B_2 * D_{m-2} + B_1 * D_{m-1} + B_0 * D_m$$

where m={0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

interrupt, interruptf, interrupts, interruptnr, interruptfnr, interruptsnr

define interrupt handling

Synopsis

```
#include <signal.h>
void (*interrupt (int sig, void(*func)(int))) (int);
void (*interruptf (int sig, void(*func)(int))) (int);
void (*interrupts (int sig, void(*func)(int))) (int);
void (*interruptnr (int sig, void(*func)(int))) (int);
void (*interruptfnr (int sig, void(*func)(int))) (int);
void (*interruptsnr (int sig, void(*func)(int))) (int);
```

Description

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every interrupt `sig`, whereas the `signal` function executes the function only once.

The `sig` argument must be one of the values that are listed in Table 3-26 on page 3-270 and the `func` argument must be one of the values that are listed in Table 3-25. The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

Table 3-25. Interrupt Handling

Func Value	Action
`SIG_DFL`	The default action is taken.
`SIG_IGN`	The signal is ignored.
Function address	The function pointed to by func is executed.

The function pointed to by `func` is executed each time the interrupt is received for handlers installed with `interrupt` (or on the first received signal in the case of handlers installed with `signal`). The `interrupt` function must be called with the `func` argument set to `SIG_IGN` to disable interrupt handling.

The `interrupt`, `interruptf`, and `interrupts` functions perform similar services and always enable nested interrupts, but they differ in the manner in which they dispatch an interrupt. The `interrupt` function is appropriate for interrupt service routines that are written in C or C++. This function uses the normal interrupt dispatcher, which provides the following services:

- Preserves all registers
- Resets the circular buffer base and length registers for linear access
- Preserves static condition flags
- Preserves the data alignment buffers
- Calls the handler function
- Restores state ready for return to the interrupted routine

When using the `interrupt` function on ADSP-TS101 processors, it takes approximately 78 cycles from the start of the interrupt dispatcher to the first instruction of the interrupt service routine (ISR) and approximately 55 cycles to return from the ISR to the interrupted code. On ADSP-TS20x processors, it takes approximately 98 cycles from the start of the interrupt dispatcher to the first instruction of the ISR and approximately 81 cycles to return from the ISR to the interrupted code.

The `interruptf` function uses a fast interrupt dispatcher that is only suitable for interrupt service routines that have been implemented in assembler. The fast interrupt dispatcher provides the same services as the normal interrupt dispatcher except that it does not preserve or restore any of the following registers:

- The circular buffer base and length registers
- The static condition flags
- The summation registers `PR 1:0`
- The data alignment buffers
- The Enhanced Communication Instruction registers

When using the `interruptf` function on ADSP-TS101 processors, it takes approximately 54 cycles from the start of the interrupt dispatcher to the first instruction of the interrupt service routine (ISR) and approximately 25 cycles to return from the ISR to the interrupted code. On ADSP-TS20x processors, it takes approximately 64 cycles from the start of the interrupt dispatcher to the first instruction of the ISR and approximately 35 cycles to return from the ISR to the interrupted code.

The `interrupts` function uses a super interrupt dispatcher that preserves and restores only the resources used in order to perform the dispatch. For this reason, it should only be used with interrupt service routines that have been written in assembler. The interrupt service routine must preserve all the registers that it uses.

When using the `interrupts` function on ADSP-TS101 processors, it takes approximately 34 cycles from the start of the interrupt dispatcher to the first instruction of the interrupt service routine (ISR) and approximately 11 cycles to return from the ISR to the interrupted code. On ADSP-TS20x processors, it takes approximately 34 cycles from the start of the interrupt dispatcher to the first instruction of the ISR and approximately 16 cycles to return from the ISR to the interrupted code.

When installing a handler for an exception condition note that the dispatcher for software exceptions performs some identification of the cause of the exception, and allows the user to install separate handlers for various classes of exception. If an illegal instruction line caused the exception, then the dispatcher will invoke the handler for the `SIGILL` signal. For a misaligned access or an access to a protected register, the handler for `SIG-`

SEGV is invoked. For a floating-point exception, the handler for the SIGFPE signal is invoked. If the exception came from another source then the handler for SIGSW is invoked. For each of these cases, if a handler has not been registered for the corresponding signal then no handler is invoked and the exception is lost.

The interrupt function returns the value of the previously installed interrupt or signal handler action.

The interrupt, interruptf, and interrupts functions install interrupt handlers that can be nested. This means that when an interrupt is being serviced by a routine installed by one of these functions, a higher priority interrupt or signal may be serviced before the routine has finished dealing with the initial interrupt.

The interruptnr, interruptfnr, and interruptsnr functions install non-reentrant interrupt handlers, meaning that on servicing an interrupt, interrupts and signals are disabled until the execution of the service routine has completely finished.

The software exception handler is a special case in the run-time library dispatchers. The 'nr' versions of interrupt are used primarily for installing hardware interrupt handlers. The software exception dispatcher does not disable hardware interrupts, even if installed by an 'nr' routine. It is necessary, therefore, to manually disable hardware interrupts around any portion of the software exception handler that must not be interrupted.

It is possible to mix both reentrant and non-reentrant interrupt and signal handlers in the same project at the expense of the additional code space used by the different dispatchers.

Error Conditions

The interrupt function returns SIG_ERR and sets errno to a positive non-zero value if it does not recognize the requested signal.

Example

```
#include <signal.h>

interrupt (SIGIRQ2, irq2_handler);
/* enable hardware interrupt pin 2 handler */

interrupt (SIGIRQ2, SIG_IGN);
/* disable hardware interrupt pin 2 handler */
```

See Also

raise, signal, signalf, signals, signalnr, signalfnr, signalsnr

localtime

convert calendar time into broken-down time

Synopsis

```
#include <time.h>
struct tm *localtime (const time_t *t);
```

Description

The `localtime` function converts a pointer to a calendar time into a broken-down time that corresponds to current time zone. A broken-down time is a structured variable, which is described in "time.h" on page 3-21. This implementation of the header file does not support the Daylight Saving flag nor does it support time zones and, thus, `localtime` is equivalent to the `gmtime` function.

The broken-down time is returned by `localtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `localtime`, or to `gmtime`.

Error Conditions

The `localtime` function does not return an error condition.

Example

```
#include <time.h>
#include <stdio.h>

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = localtime(&cal_time);
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

asctime, gmtime, mktime, time

log

natural logarithm

Synopsis

```
#include <math.h>
float logf (float x)
double log (double x)
long double logd (long double x)
```

Description

This function calculates the natural (base e) logarithm of number `x`.

If `x` equals 0, this function returns -3.4×10^{38} for a `float` return value and -1.7×10^{308} for a `long double` return value.

Algorithm

```
return = ln(x)
```

Domain

$x = [0.0 \ldots 3.4 \times 10^{38}]$ for `logf()`

$x = [0.0 \ldots 1.7 \times 10^{308}]$ for `logd()`

log10

base 10 logarithm

Synopsis

```
#include <math.h>
float log10f (float x)
double log10 (double x)
long double log10d (long double x)
```

Description

This function calculates the base 10 logarithm of number `x`.

If `x` equals 0, this function returns -3.4×10^{38} for a `float` return value and -1.7×10^{308} for a `long double` return value.

Algorithm

```
return = log(x)
```

Domain

x = [0.0 ... 3.4×10^{38}] for `log10f()`

x = [0.0 ... 1.7×10^{308}] for `log10d()`

matinv

matrix inversion

Synopsis

```
#include <matrix.h>
float *matinvf (a,n,c)
const float *a;      /* Pointer to input matrix a[][]  */
int n;               /* Number of rows in matrix a[][] */
float *c;            /* Pointer to output matrix c[][] */
```

Description

This function computes the inverse of input matrix `a[][]` and stores the result in the output matrix `c[][]`. The dimensions of matrix `a` and matrix `c` are `n` by `n`. If an inverse of the input matrix exists, a pointer to the output matrix is returned; if no inverse exists, the function returns a null pointer.

Algorithm

The function employs Gauss-Jordan elimination with full pivoting.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matmadd

matrix + matrix addition

Synopsis

```
#include <matrix.h>
void matmaddf (a,b,n,m,c)
const float *a;   /* Pointer to input matrix a[][]              */
const float *b;   /* Pointer to input matrix b[][]              */
int n;            /* Number of rows in matrix a[][] and b[][]   */
int m;            /* Number of columns in matrix a[][] and b[][] */
float *c;         /* Pointer to matrix c[][]                    */
```

Description

This function adds the input matrix `a[][]` to the input matrix `b[][]` placing the result into the output matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `m` and the dimensions of matrix `b` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} + b_{i,j}$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matmmlt

matrix * matrix multiplication

Synopsis

```
#include <matrix.h>
void matmmltf (a,n,k,b,m,c)
const float *a;         /* Pointer to input matrix a[][]     */
int n;                  /* Number of rows in matrix a[][]    */
int k;                  /* Number of columns in matrix a[][] */
const float *b;         /* Pointer to input matrix b[][]     */
int m;                  /* Number of columns in matrix b[][] */
float *c;               /* Pointer to matrix c[][]           */
```

Description

This function computes the multiplication of input matrix `a[][]` with input matrix `b[][]`, and stores the result to matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `k` and the dimensions of matrix `b` are `k` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = \sum_{l=0}^{k-1} a_{i,l} * b_{l,j}$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matmsub

matrix - matrix subtraction

Synopsis

```
#include <matrix.h>
void matmsubf (a,b,n,m,c)
const float *a;   /* Pointer to input matrix a[][]             */
const float *b;   /* Pointer to input matrix b[][]             */
int n;            /* Number of rows in matrix a[][] and b[][]  */
int m;            /* Number of columns in matrix a[][] and b[][] */
float *c;         /* Pointer to matrix c[][]                   */
```

Description

This function computes the subtraction of input matrix `a[][]` with input matrix `b[][]`, and stores the result to matrix `c[][]`.The dimensions of matrix `a[][]` are `n` and `m` and the dimensions of matrix `b` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} - b_{i,j}$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matsadd

matrix + scalar addition

Synopsis

```
#include <matrix.h>
void matsaddf (a,b,n,m,c)
const float *a;   /* Pointer to input matrix a[][]   */
float b;          /* Value of input scalar b         */
int n;            /* Number of rows in matrix a[][]  */
int m;            /* Number of columns in matrix a[][] */
float *c;         /* Pointer to matrix c[][]         */
```

Description

This function computes the addition of input matrix `a[][]` with input scalar `b`, and stores the result to matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} + b$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matsmlt

matrix * scalar multiplication

Synopsis

```
#include <matrix.h>
void matsmltf (a,b,n,m,c)
const float *a;      /* Pointer to input matrix a[][]     */
float b;             /* Value of input scalar b           */
int   n;             /* Number of rows in matrix a[][]    */
int   m;             /* Number of columns in matrix a[][] */
float *c;            /* Pointer to matrix c[][]           */
```

Description

This function computes the multiplication of input matrix `a[][]` with input scalar `b`, and stores the result to matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} * b$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matssub

matrix - scalar subtraction

Synopsis

```
#include <matrix.h>
void matssubf (a,b,n,m,c)
const float *a;      /* Pointer to input matrix a[][]     */
float b;             /* Value of input scalar b           */
int   n;             /* Number of rows in matrix a[][]    */
int   m;             /* Number of columns in matrix a[][] */
float *c;            /* Pointer to matrix c[][]           */
```

Description

This function computes the subtraction of input matrix `a[][]` with input scalar `b`, and stores the result to matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} - b$$

where i={0,1,2,...,n-1}, j={0,1,2,...,m-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

max

maximum

Synopsis

```
#include <stdlib.h>

int max (int parm1, int parm2)
long int lmax (long int parm1, long int parm2);
long long int llmax (long long int parm1, long long int parm2);
```

Description

The `max` functions return the larger of their two arguments. The functions are built-in functions that are implemented with a `MAX` instruction.

Algorithm

```
if (parm1 > parm2)
   return(parm1)
else
   return(parm2)
```

Domain

Full range for type of parameters used.

mean

mean

Synopsis

```
#include <stats.h>
float meanf (a,n)
const float a[];      /* Input vector a          */
int n;                /* Number of input samples */
```

Description

This function computes the mean of the `n` elements contained within input vector `a` and returns the result.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

$$c = \frac{1}{n} * (\sum_{i=0}^{n-1} a_i)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

min

minimum

Synopsis

```
#include <stdlib.h>

int min (int parm1, int parm2)
long int lmin (long int parm1, long int parm2);
long long int llmin (long long int parm1, long long int parm2);
```

Description

The `min` functions return the smaller of their two arguments. The functions are built-in functions that are implemented with a `MIN` instruction.

Algorithm

```
if (parm1 < parm2)
   return(parm1)
else
   return(parm2)
```

Domain

Full range for type of parameters used.

mktime

convert broken-down time into a calendar time

Synopsis

```
#include <time.h>
time_t mktime (struct tm *tm_ptr);
```

Description

The `mktime` function converts a pointer to a broken-down time, which represents a local date and time, into a calendar time. However, this implementation of `time.h` does not support either daylight saving or time zones and hence this function will interpret the argument as Greenwich Mean Time (UTC).

A broken-down time is a structured variable which is defined in the `time.h` header file as:

```
struct tm { int tm_sec;   /* seconds after the minute [0,61] */
   int tm_min;            /* minutes after the hour [0,59]   */
   int tm_hour;           /* hours after midnight [0,23]     */
   int tm_mday;           /* day of the month [1,31]         */
   int tm_mon;            /* months since January [0,11]     */
   int tm_year;           /* years since 1900                */
   int tm_wday;           /* days since Sunday [0, 6]        */
   int tm_yday;           /* days since January 1st [0,365]  */
   int tm_isdst;          /* Daylight Saving flag            */
};
```

The various components of the broken-down time are not restricted to the ranges indicated above. The `mktime` function calculates the calendar time from the specified values of the components (ignoring the initial values of `tm_wday` and `tm_yday`), and then"'normalizes" the broken-down time forcing each component into its defined range.

Error Conditions

The `mktime` function returns the value `(time_t) -1` if the calendar time cannot be represented.

Example

```
#include <time.h>
#include <stdio.h>

static const char *wday[] = {"Sun","Mon","Tue","Wed",
                             "Thu","Fri","Sat","???"};

struct tm tm_time = {0,0,0,0,0,0,0,0,0};

tm_time.tm_year = 2000 - 1900;
tm_time.tm_mday = 1;

if (mktime(&tm_time) == -1)
    tm_time.tm_wday = 7;
printf("%4d started on a %s\n",
    1900 + tm_time.tm_year,
    wday[tm_time.tm_wday]);
```

See Also

gmtime, localtime, time

modf

separate integral and fractional parts

Synopsis

```
#include <math.h>
double modf (double f, double *fraction);
float modff (float f, float *fraction);
long double modfd (long double f, long double *fraction);
```

Description

The `modf` function separates the first argument into integral and fractional portions. The fractional portion is returned and the integral portion is stored in the object pointed to by the second argument. The integral and fractional portions have the same sign as the input.

Algorithm

```
return = (|f| - floor|f|) . sign(f)

fraction = (floor|f| . sign(f))
```

Domain

$f = [-3.4 \times 10^{38} \ldots 3.4 \times 10^{38}]$ for `modff()`

$f = [-1.7 \times 10^{308} \ldots 1.7 \times 10^{308}]$ for `modfd()`

mu_compress

µ-law compression

Synopsis

```
#include <filter.h>
void mu_compress (in, out, n)
const int in[];      /* Input array                          */
int out[];           /* Output array                         */
int n;               /* Number of elements to be compressed */
```

Description

The `mu_compress` function takes a vector of linear 14-bit signed speech samples and performs µ-law compression according to ITU recommendation G.711. Each sample is compressed to 8 bits and is returned in the vector pointed to by `out`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

`C(k) = µ_law` compression of `A(k)` for k=0 to n-1.

Domain

Content of input array: –8192 to 8191

mu_expand

μ-law expansion

Synopsis

```
#include <filter.h>
void mu_expand (in, out, n)
const int in[];      /* Input array                          */
int out[];           /* Output array                         */
int n;               /* Number of elements to be expanded    */
```

Description

The `mu_expand` function inputs a vector of 8-bit compressed speech samples and expands them according to ITU recommendation G.711. Each input value is expanded to a linear 14-bit signed sample in accordance with the μ-law definition and is returned in the vector pointed to by `out`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

`C(k)``= μ_law` expansion of `A(k)` for k=0 to n-1

Domain

Content of input array: 0 to 255

norm

normalization

Synopsis

```
#include <complex.h>
complex_float normf (complex_float a);
complex_double norm (complex_double a);
complex_long_double normd (complex_long_double a);
```

Description

These functions normalize the complex input `a` and return the result.

Algorithm

$$\mathrm{Re}(c) = \frac{\mathrm{Re}(a)}{\sqrt{\mathrm{Re}^2(a) + \mathrm{Im}^2(a)}}$$

$$\mathrm{Im}(c) = \frac{\mathrm{Im}(a)}{\sqrt{\mathrm{Re}^2(a) + \mathrm{Im}^2(a)}}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `normf( )`

-1.7×10^{308} to 1.7×10^{308} for `normd( )`

perror

print an error message on standard error

Synopsis

```
#include <stdio.h>
int perror(const char *s);
```

Description

The `perror` function maps the value of the integer expression `errno` to an error message. It writes a sequence of characters to the standard error stream.

Error Conditions

The `perror` function does not return any error conditions.

Example

```
#include <stdio.h>

void test_perror(void)
{
   FILE *fp;
   fp = fopen("filedoesnotexist.txt","r");
   if (fp == NULL)
      perror("The file filedoesnotexist.txt does not exist!");
}
```

See Also

fopen

polar

construct from polar coordinates

Synopsis

```
#include <complex.h>
complex_float polarf (float mag, float phase);
complex_double polar (double mag, double phase);
complex_long_double polard (long double mag, long double phase);
```

Description

These functions transform the polar coordinate, specified by the arguments `mag` and `phase`, into a Cartesian coordinate and return the result as a complex number in which the x-axis is represented by the real part, and the y-axis by the imaginary part. The `phase` argument is interpreted as radians.

Algorithm

```
Re(c) = r*cos(θ)
Im(c) = r*sin(θ)
```

where θ is the phase, and `r` is the magnitude

Domain

phase = [–1,647,095 ... 1,647,095] for `polarf()`
mag = -3.4×10^{38} to $+3.4 \times 10^{38}$

phase = [–843,314,850 ... 843,314,850] for `polard()`
mag = -1.7×10^{308} to $+1.7 \times 10^{308}$

pow

raise to a power

Synopsis

```
#include <math.h>
float powf (float x, float y)
double pow (double x, double y)
long double powd (long double x, long double y)
```

Description

This function calculates `x` to the power `y`. If `x < 0` and `y` is not an integral value, this function returns 0. If `x = 0` and `y = 0`, this function returns 0. If overflow occurs, this function returns 3.4 x 10^{38} for a float-type return value and 1.7 x 10^{308} for a double-type return value; if underflow occurs, this function returns -3.4 x 10^{38} for a `float` return value and -1.7 x 10^{308} for a `long double` return value.

Algorithm

return = x^y

Domain

x, y = [–3.4 x 10^{38} ... 3.4 x 10^{38}]
except x<0, y ≠i, where `i` is an integer for `powf()`

x, y = [–1.7 x 10^{308} ... 1.7 x 10^{308}]
except x<0, y ≠i, where `i` is an integer for `powd()`

printf

print formatted output

Synopsis

```
#include <stdio.h>
int printf(const char *format, /* args*/ ...);
```

Description

The `printf` function places output on the standard output stream `stdout` in a form specified by `format`. The `printf` function is equivalent to `fprintf` with the `stdout` passed as the first argument. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 3-184) for a description of the valid format specifiers.

The `printf` function returns the number of characters transmitted.

Error Conditions

If the `printf` function is unsuccessful, a negative value is returned.

Example

```
#include <stdio.h>

void printf_example(void)
{
   int arg = 255;
   /* Output will be "hex:ff, octal:377, integer:255" */
   printf("hex:%x, octal:%o, integer:%d\n", arg, arg, arg);
}
```

See Also

fprintf

putc

put a character on a stream

Synopsis

```
#include <stdio.h>
int putc(int ch, char *stream);
```

Description

The `putc` function writes its argument to the output stream pointed to by `stream`, after converting `ch` from an `int` to an `unsigned char`.

If the `putc` function call is successful, `putc` returns its argument `ch`.

Error Conditions

If the call is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

void putc_example(void)
{
   /* put the character 'a' to stdout */
   if (putc('a', stdout) == EOF)
      printf("putc failed\n");
}
```

See Also

fputc

putchar

write a character to stdout

Synopsis

```
#include <stdio.h>
int putchar(int ch);
```

Description

The `putchar` function writes its argument to the standard output stream, after converting `ch` from an `int` to an `unsigned char`. A call to `putchar` is equivalent to calling `putc(ch, stdout)`.

If the `putchar` function call is successful, `putchar` returns its argument `ch`.

Error Conditions

If the `putchar` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

void putchar_example(void)
{
   /* put the character 'a' to stdout */
   if (putchar('a') == EOF)
      printf("putchar failed\n");
}
```

See Also

putc

puts

put a string to stdout

Synopsis

```
#include <stdio.h>
int puts(const char *s);
```

Description

The `puts` function writes the string pointed to by `s`, followed by a `NEWLINE` character, to the standard output stream `stdout`. The terminating null character of the string is not written to the stream.

If the function call is successful, the return value is zero or greater.

Error Conditions

The macro `EOF` is returned if `puts` was unsuccessful.

Example

```
#include <stdio.h>

void puts_example(void)
{
   /* put the string "example" to stdout */
   if (puts(“example”) < 0)
      printf("puts failed\n");
}
```

See Also

fputs

qsort

quick sort

Synopsis

```
#include <stdlib.h>
void qsort (void base, size_t nelem, size_t size,
            int (*compare) (const void *, const void *));
```

Description

The `qsort` function sorts an array of `nelem` objects, pointed to by `base`. The size of each object is specified by `size`.

The contents of the array are sorted into ascending order according to a comparison function pointed to by `compare`, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified. The `qsort` function executes a binary search operation on a pre-sorted array. Note that:

- `base` points to the start of the array
- `nelem` is the number of elements in the array
- `size` is the size of each element of the array
- `compare` points to the function used to compare two elements. It is passed two arguments that point to the array elements being compared. The function should return a value less than, equal to, or greater than zero, according to whether the first parameter is less than, equal to, or greater than the second.

Algorithm

See “Description”.

Domain

N/A

raise

force a signal

Synopsis

```
#include <signal.h>
int raise (int sig);
```

Description

The `raise` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the signals listed in priority order in Table 3-26. See the hardware reference manual for the target TigerSHARC processor for further details on the hardware interrupts.

Table 3-26. Raise Function Signals–Values and Meanings

Sig Value	Definition
SIGTIMER0LP	Timer 0 low priority interrupt
SIGTIMER1LP	Timer 1 low priority interrupt
SIGLINK0	Link port 0 interrupt
SIGLINK1	Link port 1 interrupt
SIGLINK2	Link port 2 interrupt
SIGLINK3	Link port 3 interrupt
SIGDMA0	DMA channel 0 interrupt
SIGDMA1	DMA channel 1 interrupt
SIGDMA2	DMA channel 2 interrupt
SIGDMA3	DMA channel 3 interrupt
SIGDMA4	DMA channel 4 interrupt

Table 3-26. Raise Function Signals–Values and Meanings (Cont'd)

Sig Value	Definition
SIGDMA5	DMA channel 5 interrupt
SIGDMA6	DMA channel 6 interrupt
SIGDMA7	DMA channel 7 interrupt
SIGDMA8	DMA channel 8 interrupt
SIGDMA9	DMA channel 9 interrupt
SIGDMA10	DMA channel 10 interrupt
SIGDMA11	DMA channel 11 interrupt
SIGDMA12	DMA channel 12 interrupt
SIGDMA13	DMA channel 13 interrupt
SIGIRQ0	Hardware interrupt pin 0 interrupt
SIGIRQ1	Hardware interrupt pin 1 interrupt
SIGIRQ2	Hardware interrupt pin 2 interrupt
SIGIRQ3	Hardware interrupt pin 3 interrupt
SIGBUSLK	Bus lock interrupt
SIGTIMER0HP	Timer 0 high priority interrupt
SIGTIMER1HP	Timer 1 high priority interrupt
SIGHW	Hardware error interrupt
SIGSW	Software exception
SIGDBG	Emulation debug interrupt
SIGABRT	Standard abort signal
SIGILL	Standard illegal function signal
SIGINT	Standard interactive attention signal
SIGSEGV	Standard illegal storage access signal

Table 3-26. Raise Function Signals–Values and Meanings (Cont'd)

Sig Value	Definition
SIGTERM	Standard termination request signal
SIGFPE	Standard arithmetic error signal

Error Conditions

The `raise` function returns a zero if successful, a non-zero value if it fails.

Example

```
#include <signal.h>
raise (SIGIRQ2);
       /* invoke the hardware interrupt pin 2 handler */
```

See Also

interrupt, interruptf, interrupts, interruptnr, interruptfnr, interruptsnr, signal, signalf, signals, signalnr, signalfnr, signalsnr

rand

random number generator

Synopsis

```
#include <stdlib.h>
int rand (void)
```

Description

This function returns a pseudo-random integer value in the range $[0, 2^{31} - 1]$.

For this function, the measure of randomness is its periodicity, the number of values it is likely to generate before repeating a pattern. The output of the pseudo-random number generator has a period in the order of $2^{31} - 1$.

Algorithm

The algorithm is based on a linear congruential generator.

Domain

N/A

remove

remove file

Synopsis

```
#include <stdio.h>
int remove(const char *filename);
```

Description

The `remove` function removes the file whose name is `filename`. After the function call, `filename` will no longer be accessible.

The `remove` function is supported only under the default device driver supplied by the VisualDSP++ simulator and EZ-Kit Lite system. It only operates on the host file system.

The `remove` function returns zero on successful completion.

Error Conditions

If the `remove` function is unsuccessful, a non-zero value is returned.

Example

```
#include <stdio.h>

void remove_example(char *filename)
{
   if (remove(filename))
      printf("Remove of %s failed\n", filename);
   else
      printf("File %s removed\n", filename);
}
```

See Also

rename

rename

rename a file

Synopsis

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

Description

The `rename` function will establish a new name, using the string `newname`, for a file currently known by the string `oldname`. After a successful rename, the file will no longer be accessible by `oldname`.

The `rename` function is supported only under the default device driver supplied by the VisualDSP++ simulator and EZ-Kit Lite system and it only operates on the host file system.

If `rename` is successful a value of zero is returned.

Error Conditions

If `rename` fails, the file named `oldname` is unaffected and a non-zero value is returned.

Example

```
#include <stdio.h>

void rename_file(char *new, char *old)
{
   if (rename(old, new))
      printf("rename failed for %s\n", old);
   else
      printf("%s now named %s\n", old, new);
}
```

See Also

remove

rewind

reset file position indicator in a stream

Synopsis

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

The `rewind` function sets the file position indicator for `stream` to the beginning of the file. This is equivalent to using the `fseek` routine in the following manner:

```
fseek(stream, 0, SEEK_SET);
```

The exception is that `rewind` will also clear the error indicator.

Error Conditions

The `rewind` function does not return an error condition.

Example

```
#include <stdio.h>
char buffer[20];
void rewind_example(FILE *fp)
{
   /* write "a string" to a file */
   fputs("a string", fp);
   /* rewind the file to the beginning */
   rewind(fp);
   /* read back from the file - buffer will be "a string" */
   fgets(buffer, sizeof(buffer), fp);
}
```

See Also

fseek

rfft

N point real input FFT

Synopsis

```
#include <filter.h>

void rfft (in[], t[], out[], w[], wst, n)
const float in[];            /* Input sequence           */
complex_float t[];           /* Temporary working buffer */
complex_float out[];         /* Output sequence          */
const complex_float w[];     /* Twiddle sequence         */
int wst;                     /* Twiddle factor stride    */
int n;                       /* Number of FFT points     */
```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the accelerated version of the 'Discrete Fourier Transformation' known as a 'Fast Fourier Transform' or FFT. It "decimates in frequency" by the best choice FFT algorithm, radix-4 or mixed-radix, depending on the input sequence length. At the initial stage of the transformation, the `rfft` function takes advantage of the fact that the imaginary part of the input equals zero, which in turn eliminates half of the multiplications in the butterfly.

The size of the input array `in`, the output array `out`, and the temporary working buffer `t` is `n`, where `n` represents the number of points in the FFT. If the input data can be overwritten, then the memory requirements may be reduced by specifying the input array as the output array provided that the size of the input array is at least `2*n`. Run-time performance of the function is improved if the input and output arrays are allocated in a different memory block than the twiddle table, `w`.

The twiddle table is passed in the argument `w`, which must contain at least ¾`n` complex twiddle factors. The function `twidfft` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `rfft`, then the stride factor has to be set appropriately; otherwise it should be 1. For more information, see "twidfft" on page 3-336.

The library also contains the `rfftf` function (on page 3-285), which is an optimized implementation of a fast radix-2 algorithm. The characteristics of the FFT generated by the function differ to some extent from that generated by the `rfft` function. Also, for optimal performance, the `rfftf` function makes certain assumptions concerning the alignment of the input, output, and twiddle arrays.

Algorithm

See "cfft" on page 3-104.

Domain

Input sequence length `n` must equal to either a power of two, or a power of four, and at least 16.

Example

```
/* Example to demonstrate how to generate two real FFTs using
   a single twiddle table */

#include <filter.h>

#define NDATA1 256
#define NDATA2 32

float data1[NDATA1];      /* data for a 256-point FFT */
float data2[NDATA2];      /* data for a 32-point FFT  */
```

```
complex_float output1[NDATA1];
complex_float output2[NDATA2];

static complex_float twidtab[(3*NDATA1)/4];
complex_float temp[NDATA1];
   /* note that the temporary buffer should be as large as the
      largest FFT generated */

/* Generate a twiddle table for a 256-point FFT */

twidfft (twidtab,NDATA1);
   /* note that a twiddle table is constant for a given
      number of FFT points */

/* Generate a 256-point real FFT */

rfft (data1, temp, output1, twidtab, 1, NDATA1);
   /* note that the twiddle table stride factor is 1 */

/* Generate a 32-point real FFT */

rfft (data2, temp, output2, twidtab, (NDATA1/NDATA2), NDATA2);
   /* note that the twiddle table stride factor is 8 */
```

rfft_mag

rfft magnitude

Synopsis

```
#include <filter.h>

float rfft_mag (const complex_float dm input[],
                float dm output[],
                int fftsize);
```

Description

The `rfft_mag` function computes a normalized power spectrum from the output signal generated by a `rfft` function. Due to the symmetry of a real FFT about the midpoint, this function generates the power spectrum using only the first `fftsize/2` values in the signal. The size of the signal and the size of the power spectrum is `fftsize/2`.

The Nyquist frequency is located at `fftsize/2`.

Algorithm

$$\text{magnitude}(z) = \frac{2\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

Example

```
#include <filter.h>
#define N 64

float fft_input[N];
complex_float fft_output[N];

complex_float temp[N];
complex_float twid[(3*N)/4];
```

```
float spectrum[N/2];

/* Generate a twiddle table for the FFT */

twidfft (twid, N);
    /* Note that a twiddle table is constant for a given number
       of FFT points */

/* Generate a real FFT using rfft */

rfft (fft_input, temp, fft_output, twid, 1, N);

/* Generate a power spectrum */

rfft_mag (fft_output, spectrum, N);
```

rfft2d

NxN point 2-D real input FFT

Synopsis

```
#include <filter.h>
void rfft2d (*in, *t, *out, w[], wst, n)
const float *in;           /* Pointer to input matrix a[n][n]   */
complex_float *t;          /* Pointer to working buffer t[n][n] */
complex_float *out;        /* Pointer to matrix c[n][n]         */
const complex_float w[];   /* Twiddle sequence                  */
int wst;                   /* Twiddle factor stride             */
int n;                     /* Number of FFT points              */
```

Description

This function computes a two-dimensional Fast Fourier Transform of the real input matrix `a[n][n]`, and stores the result in the complex matrix `c[n][n]`.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array provided that the memory size of the input array is at least twice `n*n`.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `rfft2d`, then the stride parameter needs to be set appropriately; otherwise, it should be one. For more information, see “twidfft” on page 3-336.

Algorithm

$$c(i,j) = \sum_{k=0}^{n-1}\sum_{l=0}^{n-1} a(k,l) * e^{-2\pi j(i*k+j*l)/n}$$

where i={0,1,...,n -1}, j={0,1,2,...,n-1}

Domain

Input sequence length `n` must equal to either a power of two, or a power of four, and at least 16.

rfftf

fast N point real input FFT

Synopsis

```
#include <filter.h>
void rfftf (in[], out[], twid[], wst, n)
const float in[];              /* Input sequence          */
complex_float out[];           /* Output sequence         */
const complex_float twid[];    /* Twiddle sequence        */
int wst;                       /* Twiddle factor stride   */
int n;                         /* Number of FFT points    */
```

Description

The `rfftf` function transforms the time domain real input signal sequence to the frequency domain by using the accelerated version of the "Discrete Fourier Transform" known as a 'Fast Fourier Transform' or FFT. It "decimates in frequency" using an optimized radix-2 algorithm.

The size of the input array `in` is `n`, where `n` represents the number of points in the FFT. The `rfftf` function has been designed for optimum performance and requires that the input array `in` be aligned on an address boundary that is a multiple of the FFT size. For certain applications, this alignment constraint may not be appropriate and, in such cases, the application should call the `rfft` function (see on page 3-278) instead (with a consequent loss of some performance).

Due to the symmetry of a real FFT, only `n/2` points of the output are computed by the function and they are stored in the output array `out`. Also, each value of the output is double the actual FFT value. This behavior is in contrast to that of the `rfft` function, which generates all `n` points of the FFT and does not double the magnitude of the FFT output.

The twiddle table is passed in the argument `twid`, which must contain at least `n/2` complex twiddle factors. The function `twidfftf` may be used to initialize the array. If the twiddle table contains more factors than required

for a particular FFT size, then the stride factor `wst` has to be set appropriately; otherwise, it should be set to 1. For more information, see "twidfftf" on page 3-338.

The twiddle tables used by the functions `rfft` and `rfftf` are not compatible. The `rfft` function uses a twiddle table that contains ¾`n` factors in which the imaginary coefficients are positive sine values, while the `rfftf` function uses a twiddle table with ½`n` factors in which the imaginary coefficients are negative sine values.

It is recommended that the twiddle table and the output array are allocated in separate memory blocks—otherwise, the performance of the function degrades.

Algorithm

See "cfftf" on page 3-111.

Domain

The number of points in the FFT must be a power of 2 and must be at least 64.

rfftf_mag

rfftf magnitude

Synopsis

```
#include <filter.h>

float rfftf_mag (const complex_float dm input[],
                 float dm output[],
                 int fftsize);
```

Description

The `rfftf_mag` function computes a normalized power spectrum from the output signal generated by a `rfftf` function. The size of the signal and the size of the power spectrum is `fftsize/2`.

The Nyquist frequency is located at `fftsize/2`.

Algorithm

$$\text{magnitude}(z) = \frac{\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

Example

```
#include <filter.h>
#define N 64

#pragma align 64
section ("data2") float fft_input[N];
section ("data1") complex_float fft_output[N/2];
section ("data1") complex_float twid_table[N/2];

float spectrum[N/2];

/* Generate a twiddle table for the FFT */
```

```
twidfftf (twid_table, N);
    /* Note that a twiddle table is constant for a given number
       of FFT points */

/* Generate a real FFT using rfftf */

rfftf (fft_input, fft_output, twid_table, 1, N);

/* Generate a power spectrum */

rfftf_mag (fft_output, spectrum, N);
```

rms

root mean square

Synopsis

```
#include <stats.h>
float rmsf (a,n)
const float a[];       /* Pointer to input vector a */
int n;                 /* Number of input samples   */
```

Description

This function computes the root mean square of the input elements contained within input vector `a` and returns the result.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

$$c = \sqrt{\frac{\sum_{i=0}^{n-1} a_i^{\,2}}{n}}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

rsqrt

reciprocal square root

Synopsis

```
#include <math.h>
float rsqrtf (float x)
double rsqrt (double x)
long double rsqrtd (long double x)
```

Description

This function calculates the reciprocal of the square root of the number `x`. If `x` is negative, the function returns 0.

Algorithm

$return = 1/(\sqrt{x})$

Domain

$x = [0.0 \text{ to } +3.4 \times 10^{38}]$ for `rsqrtf()`

$x = [0.0 \ldots 1.7 \times 10^{308}]$ for `rsqrtd()`

scanf

convert formatted input from stdin

Synopsis

```
#include <stdio.h>
int scanf(const char *format, /* args */...);
```

Description

The `scanf` function reads from the standard input stream `stdin`, interprets the inputs according to `format`, and stores the results of the conversions in it's arguments. The string pointed to by `format` contains the control format for the input with the arguments that follow being pointers to the locations where the converted results are to be written to.

The `scanf` function is equivalent to calling `fscanf` with `stdin` as its first argument. For details on the control format string, refer to "fscanf" on page 3-197.

The `scanf` function returns number of successful conversions performed.

Error Conditions

The `scanf` function returns `EOF` if it encounters an error before any conversions were performed.

Example

```
#include <stdio.h>

void scanf_example(void)
{
   short int day, month, year;
   char string[20];

   /* Scan a string from standard input */
   scanf ("%s", string);
   /* Scan a date with any separator, eg, 1-1-2006 or 1/1/2006 */
```

```
    scanf ("%hd%*c%hd%*c%hd", &day, &month, &year);
}
```

See Also

fscanf

setbuf

specify full buffering for a file or stream

Synopsis

```
#include <stdio.h>
void setbuf(FILE *stream, char* buf);
```

Description

The `setbuf` function results in the array pointed to by `buf` to be used to buffer the stream pointed to by `stream`, instead of an automatically allocated buffer. The `setbuf` function may be used only after the stream pointed to by `stream` is opened, but before it is read or written to. Note that the buffer provided must be of size `BUFSIZ` as defined in the `stdio.h` header.

If `buf` is the `NULL` pointer, the input/output will be completely unbuffered.

Error Conditions

The `setbuf` function does not return an error condition.

Example

```
#include <stdio.h>
#include <stdlib.h>
void* allocate_buffer_from_heap(FILE* fp)
{
   /* Allocate a buffer from the heap for the file pointer */
   void* buf = malloc(BUFSIZ);
   if (buf != NULL)
      setbuf(fp, buf);
   return buf;
}
```

See Also

setvbuf

setvbuf

specify buffering for a file or stream

Synopsis

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Description

The `setvbuf` function may be used after a stream has been opened, but before it is read or written to. The kind of buffering that is to be used is specified by the `type` argument. The valid values for `type` are detailed in the following table.

Type	Effect
_IOFBF	Use full buffering for output. Only output to the host system when the buffer is full, or when the stream is flushed or closed, or when a file positioning operation intervenes.
_IOLBF	Use line buffering. The buffer will be flushed whenever a `NEWLINE` is written, as well as when the buffer is full, or when input is requested.
_IONBF	Do not use any buffering at all.

If `buf` is not the `NULL` pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. Note that if `buf` is non-`NULL`, you must ensure that the associated storage continues to be

available until you close the stream identified by `stream`. The `size` argument specifies the size of the buffer required. If input/output is unbuffered, the `buf` and `size` arguments are ignored.

When the buffer contains data for a text stream (either input data or output data), the information is held in the form of 8-bit characters that are packed into 32-bit memory locations. Due to internal mechanisms used to unpack and pack this data, the I/O buffer must not reside at a memory location that is greater than the address 0x3fffffff.

If `buf` is the `NULL` pointer, buffering is enabled and a buffer of size `size` will be automatically generated.

The `setvbuf` function returns zero when successful.

Error Conditions

The `setvbuf` function will return a non-zero value if an invalid value is given for `type`, the stream has already been used to read or write data, or an I/O buffer could not be allocated.

Example

```
#include <stdio.h>

void line_buffer_stderr(void)
{
   /* stderr is not buffered - set to use line buffering */
   setvbuf (stderr,NULL,_IOLBF,BUFSIZ);
}
```

See Also

setvbuf

sign

sign

Synopsis

```
#include <math.h>
float signf (float parm1, float parm2)
double sign (double parm1, double parm2)
long double signd (long double parm1, long double parm2)
```

Description

This function copies the sign of the second argument to the first argument.

Algorithm

```
return( |parm1| * signof( parm2))
```

Domain

Full range

signal, signalf, signals, signalnr, signalfnr, signalsnr

define signal handling

Synopsis

```
#include <signal.h>
void (*signal (int sig, void(*func)(int))) (int);
void (*signalf (int sig, void(*func)(int))) (int);
void (*signals (int sig, void(*func)(int))) (int);
void (*signalnr (int sig, void(*func)(int))) (int);
void (*signalfnr (int sig, void(*func)(int))) (int);
void (*signalsnr (int sig, void(*func)(int))) (int);
```

Description

The `signal` function determines how a signal received during program execution is handled. The function sets up a handler that can respond to a single occurrence of an interrupt. It returns the value of the previously installed interrupt or signal handler action.

The `sig` argument must be one of the values that are listed in Table 3-26 on page 3-270 and the `func` argument must be one of the values that are listed in Table 3-27. The `signal` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

Table 3-27. Interrupt Handling

Func Value	Action
`SIG_DFL`	The default action is taken.
`SIG_IGN`	The signal is ignored.
Function address	The function pointed to by `func` is executed.

The function pointed to by `func` is executed once when the interrupt is received. Handling of the interrupt is then returned to the default state.

The `signal`, `signalf`, and `signals` functions perform similar services and always enable nested interrupts, but they differ in the manner in which they dispatch an interrupt. The `signal` function is appropriate for interrupt service routines that are written in C or C++. This function uses the normal interrupt dispatcher, which provides the following services:

- Preserves all registers
- Resets the circular buffer base and length registers for linear access
- Preserves static condition flags
- Preserves the data alignment buffers
- Calls the handler function
- Restores state ready for return to the interrupted routine

The `signalf` function uses a fast interrupt dispatcher that is only suitable for interrupt service routines that have been implemented in assembler. The fast interrupt dispatcher provides the same services as the normal interrupt dispatcher except that it does not preserve or restore any of the following registers:

- The circular buffer base and length registers
- The static condition flags
- The summation registers `PR 1:0`
- The data alignment buffers
- The Enhanced Communication Instruction registers

The `signals` function uses a super interrupt dispatcher that preserves and restores only the resources used in order to perform the dispatch. For this reason, it should only be used with interrupt service routines that have been written in assembler. The interrupt service routine must preserve all the registers that it uses.

When installing a handler for an exception condition note that the dispatcher for software exceptions performs some identification of the cause of the exception, and allows the user to install separate handlers for various classes of exception. If an illegal instruction line caused the exception, then the dispatcher will invoke the handler for the `SIGILL` signal. For a misaligned access or an access to a protected register, the handler for `SIGSEGV` is invoked. For a floating-point exception, the handler for the `SIGFPE` signal is invoked. If the exception came from another source then the handler for `SIGSW` is invoked. For each of these cases, if a handler has not been registered for the corresponding signal then no handler is invoked and the exception is lost.

The `signal`, `signalf`, and `signals` functions install signal handlers that can be nested. This means that when a signal is being serviced by a routine installed by one of these functions, a higher priority signal or interrupt may be serviced before the routine has finished dealing with the initial signal.

The `signalnr`, `signalfnr`, and `signalnr` functions install non-reentrant signal handlers, meaning that on servicing a signal, signals and interrupts are disabled until the execution of the service routine has completely finished.

The software exception handler is a special case in the run-time library dispatchers. The 'nr' versions of `signal` are primarily for installing hardware interrupt handlers. The software exception dispatcher does not disable hardware interrupts, even if installed by a 'nr' routine. It is necessary, therefore, to manually disable hardware interrupts around any portion of the software exception handler that must not be interrupted.

Error Conditions

The `signal` function returns `SIG_ERR` and sets `errno` to a positive non-zero value if it does not recognize the requested signal.

Example

```
#include <signal.h>

signal (SIGIRQ2, irq2_handler);
/* enable hardware interrupt pin 2 handler */

signal (SIGIRQ2, SIG_IGN);
/* disable hardware interrupt pin 2 handler */
```

See Also

interrupt, interruptf, interrupts, interruptnr, interruptfnr, interruptsnr, raise

sin

sine

Synopsis

```
#include <math.h>
float sinf (float x)
double sin (double x)
long double sind (long double x)
```

Description

The `sin` function returns the sine of the argument. The input is interpreted as a radian; the output is in the range [-1, 1]. If `x` is outside of the domain, this function returns 0.0.

Algorithm

```
return = sin(x)
```

Domain

x = [–1,647,095 ... 1,647,095] for `sinf()`

x = [–843,314,850 ... 843,314,850] for `sind()`

sinh

hyperbolic sine

Synopsis

```
#include <math.h>
float sinhf (float x)
double sinh (double x)
long double sinhd (long double x)
```

Description

The `sinh` function returns the hyperbolic sine of the argument `x`, where `x` is measured in radians. If `x` is outside the domain, this function returns 3.4×10^{38} for a `float`-type return value and 1.7×10^{308} for a `double`-type return value.

Algorithm

```
return = sinh(x)
```

Domain

$x = [-(\ln(3.4 \times 10^{38}) - \ln(2)) \ldots (\ln(3.4 \times 10^{38}) - \ln(2))]$ for `sinhf()`

$x = [-(\ln(1.7 \times 10^{308}) - \ln(2)) \ldots (\ln(1.7 \times 10^{308}) - \ln(2))]$ for `sinhd()`

snprintf

format data into an n-character array

Synopsis

```
#include <stdio.h>
int snprintf (char *str, size_t n, const char *format, ...);
```

Description

The `snprintf` function is defined in the C99 Standard (ISO/IEC 9899).

It is similar to the `sprintf` function in that `snprintf` formats data according to the argument `format`, and then writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 3-184) for a description of the valid format specifiers.

The function differs from `sprintf` in that no more than `n-1` characters are written to the output array. Any data written beyond the `n-1`th character is discarded. A terminating `NUL` character is written after the end of the last character written to the output array, unless `n` is set to zero. In that case nothing will be written to the output array and the output array may be represented by the `NULL` pointer.

The `snprintf` function returns the number of characters that would have been written to the output array `str` if `n` was sufficiently large. The return value does not include the terminating null character written to the array.

The output array will contain all of the formatted text, if the return value is not negative and is also less than `n`.

Error Conditions

The `snprintf` function returns a negative value if a formatting error occurred.

Example

```
#include <stdio.h>
#include <stdlib.h>
extern char *make_filename(char *name, int id)
{
   char *filename_template = "%s%d.dat";
   char *filename = NULL;

   int len = 0;
   int r;                 /* return value from snprintf */

   do {
      r = snprintf(filename,len,filename_template,name,id);
      if (r < 0)          /* formatting error?           */
         abort();
      if (r < len)        /* was complete string written?*/
         return filename; /* return with success         */
      filename = realloc(filename,(len=r+1));
   } while (filename != NULL);
   abort();
}
```

See Also

fprintf, sprintf, vsnprintf

sprintf

format data into a character array

Synopsis

```
#include <stdio.h>
int sprintf (char *str, const char *format, /* args */...);
```

Description

The `sprintf` function formats data according to the argument `format`, and writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 3-184) for a description of the valid format specifiers.

In all respects (other than writing to an array rather than a stream), the behavior of `sprintf` is similar to that of `fprintf`.

If the `sprintf` function is successful, it will return the number of characters written in the array, not counting the terminating `NULL` character.

Error Conditions

The `sprintf` function returns a negative value if a formatting error occurred.

Example

```
#include <stdio.h>
#include <stdlib.h>

char filename[128];

extern char *assign_filename(char *name)
{
   char *filename_template = "%s.dat";
   int r;                /* return value from sprintf */
```

```
    if ((strlen(name)+5) > sizeof(filename))
       abort();
    r = sprintf(filename, filename_template, name);
    if (r < 0)          /* sprintf failed      */
       abort();
    return filename;    /* return with success */
}
```

See Also

fprintf, snprintf

sqrt

square root

Synopsis

```
#include <math.h>
float sqrtf (float x)
double sqrt (double x)
long double sqrtd (long double x)
```

Description

The `sqrt` function returns the positive square root of the argument `x`. If `x` is negative, this function returns 0.

Algorithm

$return = \sqrt{x}$

Domain

x = [0.0 ... 3.4 x 10^{38}] for `sqrtf()`

x = [0.0 ... 1.7 x 10^{308}] for `sqrtd()`

srand

random number seed

Synopsis

```
#include <stdlib.h>
void srand (unsigned int newseed)
```

Description

The `srand` function is used to set the seed value for the `rand` function. A particular seed value always produces the same sequence of pseudo-random numbers.

Algorithm

```
generator_seed = newseed
```

Domain

0 to `RAND_MAX`

sscanf

convert formatted input in a string

Synopsis

```
#include <stdio.h>
int sscanf(const char *s, const char *format, /* args */...);
```

Description

The `sscanf` function reads from the string `s`. The function is equivalent to `fscanf`, with the exception of the string being read from a string rather than a stream. The behavior of `sscanf` when reaching the end of the string equates to `fscanf` reaching the `EOF` in a stream. For details on the control format string, refer to "fscanf" on page 3-197.

The `sscanf` function returns the number of items successfully read.

Error Conditions

If the `sscanf` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

void sscanf_example(const char *input)
{
   short int day, month, year;
   char string[20];

   /* Scan for a string from "input" */
   sscanf (input, "%s", string);
   /* Scan a date with any separator, eg, 1-1-2006 or 1/1/2006 */
   sscanf (input, "%hd%*c%hd%*c%hd", &day, &month, &year);
}
```

See Also

fscanf

strftime

format a broken-down time

Synopsis

```
#include <time.h>
size_t strftime (char *buf,
                 size_t buf_size,
                 const char *format,
                 const struct tm *tm_ptr);
```

Description

The `strftime` function formats the broken-down time `tm_ptr` into the `char` array pointed to by `buf`, under the control of the format string `format`. `buf_size` characters (including the null terminating character) are written to `buf`.

Similar to `printf`, the `strftime` format string consists of ordinary characters, which are copied unchanged to the `char` array `buf`, and zero or more conversion specifiers. A conversion specifier starts with the character `%` and is followed by a character that indicates the form of transformation required – the supported transformations are given below in Table 3-28. The `strftime` function only supports the "C" locale, and this is reflected in the table.

Table 3-28. Conversion Specifiers Supported by `strftime`

Conversion Specifier	Transformation	ISO/IEC 9899
`%a`	abbreviated weekday name	yes
`%A`	full weekday name	yes
`%b`	abbreviated month name	yes
`%B`	full month name	yes
`%c`	date and time presentation in the form of `DDD MMM dd hh:mm:ss yyyy`	yes

Table 3-28. Conversion Specifiers Supported by `strftime` (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
`%C`	century of the year	POSIX.2-1992 + ISO C99
`%d`	day of the month (`01 - 31`)	yes
`%D`	date represented as `mm/dd/yy`	POSIX.2-1992 + ISO C99
`%e`	day of the month, padded with a space character (cf `%d`)	POSIX.2-1992 + ISO C99
`%F`	date represented as `yyyy-mm-dd`	POSIX.2-1992 + ISO C99
`%h`	abbreviated name of the month (same as `%b`)	POSIX.2-1992 + ISO C99
`%H`	hour of the day as a 24-hour clock (`00-23`)	yes
`%I`	hour of the day as a 12-hour clock (`00-12`)	yes
`%j`	day of the year (`001-366`)	yes
`%k`	hour of the day as a 24-hour clock padded with a space (`0-23`)	no
`%l`	hour of the day as a 12-hour clock padded with a space (`0-12`)	no
`%m`	month of the year (`01-12`)	yes
`%m`	month of the year (`01-12`)	yes
`%M`	minute of the hour (`00-59`)	yes
`%n`	newline character	POSIX.2-1992 + ISO C99
`%p`	`AM` or `PM`	yes
`%P`	`am` or `pm`	no
`%r`	time presented as either `hh:mm:ss AM` or as `hh:mm:ss PM`	POSIX.2-1992 + ISO C99
`%R`	time presented as `hh:mm`	POSIX.2-1992 + ISO C99
`%S`	second of the minute (`00-61`)	yes

Table 3-28. Conversion Specifiers Supported by strftime (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
%t	tab character	POSIX.2-1992 + ISO C99
%T	time formatted as %H:%M:%S	POSIX.2-1992 + ISO C99
%U	week number of the year (week starts on Sunday) (00-53)	yes
%w	weekday as a decimal (0-6) (0 if Sunday)	yes
%W	week number of the year (week starts on Sunday) (00-53)	yes
%x	date represented as mm/dd/yy (same as %D)	yes
%X	time represented as hh:mm:ss	yes
%y	year without the century (00-99)	yes
%Y	year with the century (nnnn)	yes
%Z	the time zone name, or nothing if the name cannot be determined	yes
%%	% character	yes

The current implementation of time.h does not support time zones and, therefore, the %Z specifier does not generate any characters.

The strftime function returns the number of characters (not including the terminating null character) that have been written to buf.

Error Conditions

The strftime function returns zero if more than buf_size characters are required to process the format string. In this case, the contents of the array buf will be indeterminate.

Example

```
#include <time.h>
#include <stdio.h>

extern void
print_time(time_t tod)
{
   char tod_string[100];

   strftime(tod_string,
            100,
            "It is %M min and %S secs after %l o'clock (%p)",
            gmtime(&tod));
   puts(tod_string);
}
```

See Also

ctime, gmtime, localtime, mktime

strtod

convert string to double

Synopsis

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr)
```

Description

The `strtod` function converts a character string to a `double` value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument is a pointer to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:.

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan. If `endptr` is not `NULL`, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Domain

The number should fit within the dynamic range of a `double`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

strtof

convert string to float

Synopsis

```
#include <stdlib.h>
float strtof (const char *string, char **end)
```

Description

The `strtof` function converts a character string to a single-precision floating-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument is a pointer to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:.

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan. If `endptr` is not `NULL`, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Domain

The number should fit within the dynamic range of a `float`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `FLT_MAX` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

strtoi

convert string to integer

Synopsis

```
#include <stdlib.h>
int strtoi (const char *string, char **end, int base)
```

Description

The `strtoi` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *`whitespace`* can consist of spaces and/or tab characters, *`sign`* is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `f` for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets `*end` to that location, provided that `end` is not a null pointer.

If the third argument *`base`* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *`base`* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *`base`* does not correspond to any of the above values, it is set to 10. If the third argument *`base`* is between 11 and 36 (inclusively) the letters of the alphabet from `a` to `z` are used as needed to represent the extra digits in the corresponding base.

Domain

-2,147,483,648 to 2,147,483,647

strtol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long strtol (const char *string, char **end, int base)
```

Description

The `strtol` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *`whitespace`* can consist of spaces and/or tab characters, *`sign`* is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `f` for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets `*end` to that location, provided that `end` is not a null pointer.

If the third argument *`base`* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *`base`* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *`base`* does not correspond to any of the above values, it is set to 10. If the third argument *`base`* is between 11 and 36 (inclusively), the letters of the alphabet from `a` to `z` are used as needed to represent the extra digits in the corresponding base.

Domain

-2,147,483,648 to 2,147,483,647

strtold

convert string to long double

Synopsis

```
#include <stdlib.h>
long double strtold (const char *string, char **end)
```

Description

The `strtold` function converts a character string to a double-precision floating-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [digits] [.digits] [{e|E} [sign] digits]
```

where *`whitespace`* can consist of spaces and/or tab characters, *`sign`* is either plus (`+`) or minus (`-`), and *`digits`* are one or more decimal digits. If no digits appear before the decimal point then at least one must appear after the decimal point. The decimal digits may be followed by an exponent denoted by one of the following characters: `e` or `E`, followed by a decimal integer which may be signed.

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets `*end` to that location, provided that `end` is not a null pointer.

Domain

The number should fit within the dynamic range of a `long double`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate)

LDBL_MAX is returned. If the correct value results in an underflow, 0.0 is returned. The ERANGE value is stored in errno in the case of either an overflow or underflow.

strtoll

convert string to long long integer

Synopsis

```
#include <stdlib.h>
long long strtoll (const char *string, char **end, int base)
```

Description

The `strtoll` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *`whitespace`* can consist of spaces and/or tab characters, *`sign`* is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `f` for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets `*end` to that location, provided that `end` is not a null pointer.

If the third argument *`base`* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *`base`* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *`base`* does not correspond to any of the above values, it is set to 10. If the third argument *`base`* is between 11 and 36 (inclusively), the letters of the alphabet from `a` to `z` are used as needed to represent the extra digits in the corresponding base.

Domain

–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

strtoul

convert string to unsigned long integer

Synopsis

```
#include <stdlib.h>
unsigned long strtoul (const char *string, char **end, int base)
```

Description

The `strtoul` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *`whitespace`* can consist of spaces and/or tab characters, `sign` is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `z` for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets `*end` to that location, provided that `end` is not a null pointer.

If the third argument *`base`* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *`base`* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *`base`* does not correspond to any of the above values, it is set to 10. If the third argument *`base`* is between 11 and 36 (inclusively), the letters of the alphabet from `a` to `z` are used as needed to represent the extra digits in the corresponding base.

Domain

0 to 4,294,967,295

strtoull

convert string to unsigned long long integer

Synopsis

```
#include <stdlib.h>
unsigned long long strtoull (const char *string, char **end,
                             int base);
```

Description

The `strtoull` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *`whitespace`* can consist of spaces and/or tab characters, `sign` is either plus (`+`) or minus (`-`), *`base`* is `0` for octal and `0x` for hexadecimal, and *`digits`* are one or more decimal digits (or letters from `a` to `z` for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets `*end` to that location, provided that `end` is not a null pointer.

If the third argument *`base`* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *`base`* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *`base`* does not correspond to any of the above values, it is set to 10. If the third argument *`base`* is between 11 and 36 (inclusively), the letters of the alphabet from `a` to `z` are used as needed to represent the extra digits in the corresponding base.

Domain

0 to 18,446,744,073,709,551,615

tan

tangent

Synopsis

```
#include <math.h>
float tanf (float x)
double tan (double x)
long double tand (long double x)
```

Description

The `tan` function returns the tangent of the argument `x`, where `x` is measured in radians. The library function returns 0 for any input argument that is outside the defined domain.

Algorithm

```
return = tan(x)
```

Domain

x = [–6,588,397 ... 6,588,397] for `tanf()`

x = [–421,657,424 ... 421,657,424] for `tand()`

tanh

hyperbolic tangent

Synopsis

```
#include <math.h>
float tanhf (float x)
double tanh (double x)
long double tanhd (long double x)
```

Description

This function calculates the hyperbolic tangent of number `x`, where `x` is measured in radians. If `x` is outside the domain, this function returns 3.4×10^{38} for a float-type return value and 1.7×10^{308} for a double-type return value.

Algorithm

```
return = tanh(x)
```

Domain

$x = [-(\ln(3.4 \times 10^{38}) - \ln(2)) \ldots (\ln(3.4 \times 10^{38}) - \ln(2))]$ for `tanhf()`

$x = [-(\ln(1.7 \times 10^{308}) - \ln(2)) \ldots (\ln(1.7 \times 10^{308}) - \ln(2))]$ for `tanhd()`

time

calendar time

Synopsis

```
#include <time.h>
time_t time (time_t *t);
```

Description

The `time` function returns the current calendar time which measures the number of seconds that have elapsed since the start of a known epoch. As the calendar time cannot be determined in this implementation of `time.h`, a result of `(time_t) -1` is returned. The function's result is also assigned to its argument, if the pointer to `t` is not a null pointer.

Error Conditions

The `time` function will return the value `(time_t) -1` if the calendar time is not available.

Example

```
#include <time.h>
#include <stdio.h>

if (time(NULL) == (time_t) -1)
    printf("Calendar time is not available\n");
```

See Also

ctime, gmtime, localtime

transpm

matrix transpose

Synopsis

```
#include <matrix.h>
void transpmf (a,n,m,c)
const float *a;      /* Pointer to input matrix a[][]     */
int n;               /* Number of rows in matrix a[][]    */
int m;               /* Number of columns in matrix a[][] */
float *c;            /* Pointer to output matrix c[][]    */
```

Description

This function computes the linear algebraic transpose of input matrix `a[][]` and stores the result in `c[][]`. The dimensions of matrix `a` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `m` and `n`.

Algorithm

$c_{ji} = a_{ij}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

twidfft

generate FFT twiddle factors

Synopsis

```
#include <filter.h>
void twidfft (w[], n)
complex_float w[];      /* Twiddle sequence     */
int n;                  /* Number of FFT points */
```

Description

Various different versions of the FFT are provided, including `cfft`, `rfft`, `ifft`, `cfft2d`, `rfft2d`, `ifft2d`. The number of points is provided as an argument; when appropriate, the library uses radix-2 or radix-4 implementations.

For efficiency, the twiddle table is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. The function `twidfft` is the initialization function.

Several FFTs of different sizes can all be accommodated with the same twiddle factor table. Simply allocate the table at the maximum size. Each FFT has an additional parameter, the “stride” of the twiddle table. To use the whole table, specify a stride of 1. If your FFT uses only half the points of the largest, the stride should be 2 (this takes only every other element).

The twiddle table generated by the `twidfft` function is not compatible with the twiddle table generated by the function `twidfftf`, and must not be used in conjunction with the fast FFT functions `cfftf` and `rfftf`.

Algorithm

This function takes FFT length `n` as an input parameter and generates the lookup table of complex twiddle coefficients. 3/4 of one period of sine/cosine is described by 3/4 `n` complex samples in the lookup table.

The samples are generated as follows:

$$twid_re(k) = \cos\left(\frac{2\pi}{n}k\right)$$

$$twid_im(k) = \sin\left(\frac{2\pi}{n}k\right)$$

where `k` = {0,1,2,..., ¾n – 1}

Domain

The `n` parameter must be either a power of two, or a power of four, and at least 16.

twidfftf

generate FFT twiddle factors for a fast FFT

Synopsis

```
#include <filter.h>
void twidfftf (w[], n)
complex_float w[];      /* Twiddle sequence     */
int n;                  /* Number of FFT points */
```

Description

The `twidfftf` function generates complex twiddle factors for the fast FFT functions `cfftf` and `rfftf`, and stores the coefficients in the vector `w`. The vector `w`, known as the twiddle table, is normally calculated once and is then passed to a fast FFT as a separate argument. The size of the table must be ½ of `n`, the number of points in the FFT.

The same twiddle table may be used to calculate FFTs of different sizes provided that the table was generated for the largest FFT. Each FFT function has a stride parameter that the function uses to stride through the twiddle table. Normally, this stride parameter is set to 1, but to generate a smaller FFT, the argument should be scaled appropriately. For example, if a twiddle table was generated for an FFT with `N` points, then the same twiddle table may be used to generate a `N/2`-point FFT provided that the stride parameter is set to 2, or a `N/4`-point FFT if the parameter is set to 4.

The twiddle table generated by the `twidfftf` function is not compatible with the twiddle table generated by the `twidfft` function.

Algorithm

The function calculates a lookup table of complex twiddle factors. The coefficients generated are:

$$twid_re(k) = \cos\left(\frac{2\pi}{n}k\right)$$

$$twid_im(k) = -\sin\left(\frac{2\pi}{n}k\right)$$

where k = {0, 1, 2, …, n /2 – 1}

Domain

The number of points in the FFT must be a power of 2 and must be at least 32.

ungetc

push character back into input stream

Synopsis

```
#include <stdio.h>
int ungetc(int uc, FILE *stream);
```

Description

The `ungetc` function pushes the character specified by `uc` back onto `stream`. The unsigned chars that have been pushed back onto `stream` will be returned by any subsequent read of `stream` in the reverse order of their pushing.

A successful call to the `ungetc` function will clear the `EOF` indicator for `stream`. The file position indicator for `stream` is decremented for every successful call to `ungetc`.

Upon successful completion, `ungetc` returns the character pushed back after conversion.

Error Conditions

If the `ungetc` function is unsuccessful, `EOF` is returned.

Example

```
#include <stdio.h>

void ungetc_example(FILE *fp)
{
   int ch, ret_ch;
   /* get char from file pointer */
   ch = fgetc(fp);
   /* unget the char, return value should be char */
   if ((ret_ch = ungetc(ch, fp)) != ch)
      printf("ungetc failed\n");
```

```
    /* make sure that the char had been placed in the file */
    if ((ret_ch = fgetc(fp)) != ch)
       printf("ungetc failed to put back the char\n");
}
```

See Also

fseek, fsetpos, getc

var

variance

Synopsis

```
#include <stats.h>
float varf (a,n)
const float a[];      /* Pointer to input vector a */
int n;                /* Number of input samples   */
```

Description

This function computes the variance of the input elements contained within input vector `a` and returns the result.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

$$c = \frac{n * \sum_{i=0}^{n-1} a_i^{\,2} - (\sum_{i=0}^{n-1} a_i)^2}{n(n-1)}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecdot

real vector dot product

Synopsis

```
#include <vector.h>
float vecdotf (a,b,n)
const float a[];       /* Input vector a */
const float b[];       /* Input vector b */
int n;                 /* Element count  */
```

Description

This function computes the dot product of the two input vectors `a` and `b`, and returns the scalar result.

Algorithm

$$return = \sum_{i=0}^{n-1} a_i * b_i$$

where `i` = {0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecsadd

real vector + scalar addition

Synopsis

```
#include <vector.h>
void vecsaddf (a,b,c,n)
const float a[];         /* Input vector a */
float b;                 /* Input scalar   */
float c[];               /* Output vector  */
int n;                   /* Element count  */
```

Description

This function adds input scalar `b` to each element of input vector `a`. The results are stored in the output vector `c`.

Algorithm

$c_i = a_i + b$

where `i` = {0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecsmlt

real vector * scalar multiplication

Synopsis

```
#include <vector.h>
void vecsmltf (a,b,c,n)
const float a[];          /* Input vector a */
float b;                  /* Input scalar   */
float c[];                /* Output vector  */
int n;                    /* Element count  */
```

Description

This function multiplies each element of input vector `a` by input scalar `b`. The results are stored in the output vector `c`.

Algorithm

$c_i = a_i * b$

where $i = \{0,1,2,...,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecssub

real vector - scalar subtraction

Synopsis

```
#include <vector.h>
void vecssubf (a,b,c,n)
const float a[];          /* Input vector a */
float b;                  /* Input scalar   */
float c[];                /* Output vector  */
int n;                    /* Element count  */
```

Description

This function subtracts input scalar `b` from each element of input vector `a`. The results are stored in the output vector `c`.

Algorithm

$c_i = a_i - b$

where `i` = {0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecvadd

real vector + vector addition

Synopsis

```
#include <vector.h>
void vecvaddf (a,b,c,n)
const float a[];          /* Input vector a */
const float b[];          /* Input vector b */
float c[];                /* Output vector  */
int n;                    /* Element count  */
```

Description

This function adds two input vectors. The results are stored in the output vector `c`.

Algorithm

$c_i = a_i + b_i$

where `i` = {0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecvmlt

real vector * vector multiplication

Synopsis

```
#include <vector.h>
void vecvmltf (a,b,c,n)
const float a[];          /* Input vector a */
const float b[];          /* Input vector b */
float c[];                /* Output vector  */
int n;                    /* Element count  */
```

Description

This function multiplies two input vectors `a` and `b`. The results are stored in the output vector `c`.

Algorithm

$c_i = a_i * b_i$

where `i` = {0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecvsub

real vector - vector subtraction

Synopsis

```
#include <vector.h>
void vecvsubf (a,b,c,n)
const float a[];          /* Input vector a */
const float b[];          /* Input vector b */
float c[];                /* Output vector  */
int n;                    /* Element count  */
```

Description

This function subtracts input vector `b` from input vector `a`. The results are stored in the output vector `c`.

Algorithm

$c_i = a_i - b_i$

where `i` = {0,1,2,...,n-1}

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vfprintf

print formatted output of a variable argument list

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vfprintf(FILE *stream, const char *format, va_list ap);
```

Description

The `vfprintf` function formats data according to the argument `format`, and then writes the output to the stream `stream`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters that are used to control how the data is formatted. Refer to `fprintf` (on page 3-184) for a description of the valid format specifiers.

The `vfprintf` function behaves in the same manner as `fprintf`, except that instead of being a function taking a variable number or arguments, it is called with an argument list `ap` of type `va_list` (as defined in `stdarg.h`).

If the `vfprintf` function is successful, it will return the number of characters output.

Error Conditions

The `vfprintf` function returns a negative value if unsuccessful.

Example

```
#include <stdio.h>
#include <stdarg.h>

void write_name_to_file(FILE *fp, char *name_template, ...)
{
   va_list p_vargs;
   int ret;                    /* return value from vfprintf */

   va_start (p_vargs,name_template);
```

```
    ret = vfprintf(fp, name_template, p_vargs);
    va_end (p_vargs);

    if (ret < 0)
       printf("vfprintf failed\n");
}
```

See Also

fprintf

vprintf

print formatted output of a variable argument list to stdout

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vprintf(const char *format, va_list ap);
```

Description

The `vprintf` function formats data according to the argument `format`, and then writes the output to the standard output stream `stdout`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters used to control how the data is formatted. Refer to `fprintf` (on page 3-184) for a description of the valid format specifiers.

The `vprintf` function behaves in the same manner as `vfprintf`, with `stdout` provided as the pointer to the stream.

If the `vprintf` function is successful, it will return the number of characters output.

Error Conditions

The `vprintf` function returns a negative value if unsuccessful.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void print_message(int error, char *format, ...)
{
   /* This function is called with the same arguments as for */
   /* printf but if the argument error is not zero, then the */
   /* output will be preceded by the text "ERROR:"           */
```

```
    va_list p_vargs;
    int ret;                    /* return value from vprintf */

    va_start (p_vargs, format);
    if (!error)
        printf("ERROR: ");
    ret = vprintf(format, p_vargs);
    va_end (p_vargs);

    if (ret < 0)
       printf("vprintf failed\n");
}
```

See Also

fprintf, vprintf

vsnprintf

format argument list into an n-character array

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vsnprintf (char *str, size_t n, const char *format,
               va_list args);
```

Description

The `vsnprintf` function is similar to the `vsprintf` function in that it formats the variable argument list `args` according to the argument `format`, and writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters used to control how the data is formatted. Refer to `fprintf` (on page 3-184) for a description of the valid format specifiers.

The function differs from `vsprintf` in that no more than `n-1` characters are written to the output array. Any data written beyond the `n-1`th character is discarded. A terminating `NUL` character is written after the end of the last character written to the output array, unless `n` is set to zero. In that case nothing will be written to the output array and the output array may be represented by the `NULL` pointer.

The `vsnprintf` function returns the number of characters that would have been written to the output array `str` if `n` was sufficiently large. The return value does not include the terminating `NUL` character written to the array.

Error Conditions

The `vsnprintf` function returns a negative value if unsuccessful.

Example

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <stdarg.h>

char *message(char *format, ...)
{
   char *message = NULL;
   int len = 0;
   int r;
   va_list p_vargs;        /* return value from vsnprintf */

   do {
      va_start (p_vargs,format);
      r = vsnprintf (message,len,format,p_vargs);
      va_end (p_vargs);
      if (r < 0)           /* formatting error?           */
         abort();
      if (r < len)         /* was complete string written?*/
         return message;   /* return with success         */
      message = realloc (message,(len=r+1));
   } while (message != NULL);
   abort();
}
```

See Also

fprintf, snprintf

vsprintf

format argument list into a character array

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vsprintf (char *str, const char *format, va_list args);
```

Description

The `vsprintf` function formats the variable argument list `args` according to the argument `format`, and writes the output to the array `str`. The argument `format` contains a set of conversion specifiers, directives, and ordinary characters used to control how the data is formatted. Refer to `fprintf` (on page 3-184) for a description of the valid format specifiers.

The `vsprintf` function behaves in the same manner as `sprintf`. The exception is that instead of taking a variable number or arguments function, it is called with an argument list `args` of type `va_list` (as defined in `stdarg.h`.)

The `vsprintf` function returns the number of characters that have been written to the output array `str`. The return value does not include the terminating `NUL` character written to the array.

Error Conditions

The `vsprintf` function returns a negative value if unsuccessful.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

char filename[128];
```

```
char *assign_filename(char *filename_template, ...)
{
   char *message = NULL;

   int r;
   va_list p_vargs;        /* return value from vsprintf */

   va_start (p_vargs,filename_template);
   r = vsprintf(&filename[0], filename_template, p_vargs);
   va_end (p_vargs);
   if (r < 0)              /* formatting error?          */
      abort();

   return &filename[0];    /* return with success        */
}
```

See Also

fprintf, sprintf, snprintf

zero_cross

count zero crossings

Synopsis

```
#include <stats.h>
int zero_crossf (a,n)
const float a[];        /* Pointer to input vector a */
int n;                  /* Number of input samples   */
```

Description

This function computes the number of times that a signal crosses over the zero line and returns the result. If all the input values are either positive or zero, or they are all either negative or zero, then the function returns a zero.

There are constraints in the use of this function.For more information, see “stats.h – Statistical Functions” on page 3-27.

Algorithm

The actual algorithm is different from the one shown below because the algorithm needs to handle the case where an element of the array is zero. However, this example should give you an understanding of the algorithm.

```
if (a(i) > 0 && a(i+1) < 0 ) || (a(i) < 0 && a(i+1) > 0)

   Number of zeros is increased by one.
```

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

I INDEX

Symbols

µ-Law companders, 3-24
µ-Law compression, 3-258
µ-Law expansion, 3-259

Numerics

128-bit data types, 1-115
16-bit data types, 1-131
2-D convolution, 3-132
32-bit, data types, 1-77, 1-140
4-word boundary, 1-129
64-bit
  data types, 1-77
  floating-point arithmetic, 1-79
  integer support, 1-114

A

-A (assert) compiler switch, 1-23
Abridged C++ Library
  overview, 3-2
  support, 3-32
abs (absolute value) function, 3-72
accums pointer, 1-177
a_compress (A-law compression) function, 3-70
acos (arc cosine) function, 3-73
addbitrev (bit-reversed adder) function, 3-74
add_devtab_entry function, 3-55
__ADI_LIBEH__ macro, 1-73
__ADI_THREADS macro, 1-64
__ADSPTS101__ macro, 1-247
__ADSPTS201__ macro, 1-247
__ADSPTS202__ macro, 1-247
__ADSPTS203__ macro, 1-247
__ADSPTS20x__ macro, 1-247
__ADSPTS__ macro, 1-247
a_expand (A-law expansion) function, 3-71
A-law companders, 3-24
algorithm header file, 3-36
alias, avoiding, 2-20
-align-branch-lines compiler switch, 1-25
alignment inquiry keyword, 1-243
align num pragma, 1-182
__alignof__ keyword, 1-243
alog10 functions, 3-77
alternate heap interface
  functions, 1-269
alternate heap interface, functions, 1-269
-alttok (alternative tokens) C++ mode compiler switch, 1-25
-always-inline compiler switch, 1-26
-always-inline switch, 1-96
-anach (enable C++ anachronisms) compiler switch, 1-71
anachronisms
  default C++ mode, 1-71
  disabling in C++ mode, 1-75
__ANALOG_EXTENSIONS__ macro, 1-247
-annotate-loop-instr compiler switch, 1-27

annotations
  assembly code, 2-73
  assembly source code position, 2-84
  defined, 2-7
  loop identification, 2-79
  modulo scheduling, 2-92, 2-93
  vectorization, 2-90
ANSI standard
  ccts compiler, 3-1
  -extra-keywords switch, 1-34
anti-log, functions, 3-75
argc support, 1-264
arg (get phase of complex number) functions, 3-79
arguments
  handling, 1-259
  number, 1-261
  passing, 1-259
  words, 1-259
argv/argc arguments, 1-264
__argv_string variable, 1-264
argv support, 1-264
arithmetic
  mixed mode operations, 1-236
  operators, fractional, 1-234
  operators, int2x16, 1-133
arithmetic operators, int4x16, 1-138
array
  length, 1-123
  placing in memory, 2-25
  zero length, 1-241
ASCII string, *see* atof, atoi, atol functions
asctime (convert broken-down time into a string) function, 3-80
asctime function, 3-142
asin (arc sine) function, 3-82
asm
  compiler keyword, 1-88, 1-100
  keyword, 1-244
  statement, 1-242, 2-24
asm()
  construct, 1-100
  constructs flow control, 1-113
  operand constraints, 1-109
asm compiler keyword, *see also* inline assembly language support keyword (asm)
assembly
  annotations, 2-7
  code annotations, 2-73
  construct, reordering and optimization, 1-111
  construct, template, 1-102
  construct, with input and output operands, 1-112
  construct operand, 1-105
  constructs, with multiple instructions, 1-110
  subroutines, 1-280
assembly language subroutine, calling, 1-280
assert.h header file, 3-11
atan2 (arc tangent of quotient) function, 3-84
atan (arc tangent) function, 3-83
atof (convert string to double) function, 3-85
atoi (convert string to integer) function, 3-87
atol (convert string to long integer) function, 3-88
atold (convert string to long double) function, 3-89
atoll (convert string to long long integer) function, 3-91
__attribute__ keyword, 1-244
attributes
  file, 1-27, 1-34, 1-44
  functions, variables and types, 1-244
-auto-attrs switch, 1-27

autocoh (autocoherence) function, 3-92
autocorr (autocorrelation) function, 3-93
automatic
  inlining, 1-49, 1-83, 1-95, 2-23
  loop control variables, 2-36
  variables, 1-115
avg (mean of two values) functions, 3-94

B

bank, qualifier, 1-118, 2-28, 2-53
base 10, anti-log functions, 3-77
basic arithmetic functions, 3-23
basic cycle counting, 3-40
binary array search, *see* bsearch function
bitfields
  signed, 1-62
  unsigned, 1-65
Bit FIFO temporaries register, 1-278
BITS_PER_WORD constant, 3-60
bitwise logical operators, int2x16 values, 1-134
bool, *see* Boolean type support keywords (bool, true, false)
Boolean type keywords (bool, true, false), 1-88, 1-121
broken-down time, 3-21, 3-240, 3-312
bsearch (binary search in sorted array) function, 3-95
-bss (placing data in bsz) compiler switch, 1-27
buf field, 3-63
-build-lib (build library) compiler switch, 1-27
__builtin_aligned function, 1-129, 2-13, 2-18, 2-51
__builtin_circindex function, 2-42
__builtin_circptr function, 2-42, 2-43
built-in functions
  builtins.h, 1-127
  circular buffer, 1-142
  circular buffer data alignment buffer (DAB), 1-162
  communications logic unit operation, 1-164
  data alignment buffer (DAB), 1-161
  defined, 1-126
  instructions generated by, 1-145
  mapping to machine instructions, 1-146
  math, 1-143
  optimization guidance, 1-129
  system, 2-41
__builtin_ prefix, 1-145
__builtin_quad extended type, 1-79, 1-115
builtins.h header file, 1-127, 1-128, 1-145
__builtin_sysreg_read built-in function, 1-128
__builtin_sysreg_write built-in function, 1-128
byte-addressing mode
  byte sizes, 1-91
  compiler flag for, 1-91
  data pointer, 1-161
  part-words, 1-93
  pointers, 1-92
  pragma used in, 1-93, 1-181
  selecting, 1-4, 1-28
  sizeof operator, 1-91
  supporting libraries, 1-94
  __TS_BYTE_ADDRESS macro, 1-91

C

C++
  Abridged Library, 3-32
  class constructor functions, 1-60, 1-120
  embedded library, 3-38
  exception handling, 1-22
  exception handling support library, 3-5
  exit routine, 3-6
  fractional type support, 1-234
  gcc compatibility features not supported, 1-237
  header files, for C library facilities, 3-35
  template inclusion control pragma, 1-222
  virtual lookup tables, 1-60, 1-120
-c89 (ISO/IEC 9899 1990 standard) compiler switch, 1-22
cabs (complex absolute value) function, 3-97
cadd (complex addition) function, 3-98
calendar time, 3-21, 3-334
callee preserved registers, 1-273
caller save registers, 1-274
calloc function, 1-268
cartesian (Cartesian to polar) functions, 3-99
cassert header, 3-36
C/C++
  code optimization, 2-2
  run-time environment, 1-252
C/C++ language extensions
  inline, 1-95
  inline keyword, 1-95
C/C++ library functions, calling, 3-4
-c++ (C++ mode) compiler switch, 1-22
C/C++ mode selection switches
  -c89, 1-22
  -c++ (C++ mode), 1-22
CCNT0 register, 3-49
CCNT1 register, 3-49
-C (comments) compiler switch, 1-27
-c (compile only) compiler switch, 1-28
C/C++ run-time, library files, 3-5
cctype header, 3-36
cdiv (complex division) function, 3-101
ceil (ceiling) function, 3-102
cerrno header, 3-36
C exit routine, 3-6
cexp (complex exponential) function, 3-103
cfft2d (NxN point 2-d complex input FFT) function, 3-109
cfftf (fast N point complex input FFT) function, 3-111
cfft_mag function, 3-107
cfft magnitude, 3-107
cfftN (N-point complex input fast Fourier transform) functions, 3-281, 3-287
cfft (N point complex input FFT) function, 3-104
cfloat header, 3-36
CHAR32 qualifier, 1-214
CHAR8 qualifier, 1-214
CHARANY qualifier, 1-214
-char-size-8|32 compiler switch, 1-28
-char-size-any compiler switch, 1-28
-check-init-order compiler switch, 1-72
circular buffer
  built-in functions, 1-142
  DAB intrinsics, 1-162
  increment of index, 1-142
  increment of pointer, 1-142
  operation on array, 1-143
  switch setting, 1-35
  used in DSP-style code, 2-42

C language extensions
asm keyword, 1-100
bool keyword, 1-121
compound statements, 1-123
C++ style comments, 1-90
false keyword, 1-88, 1-121
indexed initializers, 1-125
list, 1-88
non-constant initializers, 1-124
segment keyword, 1-88
true keyword, 1-88, 1-121
class conversion optimization pragmas, 1-216
clearerr, 3-113
climits header, 3-36
clip (clip) functions, 3-114
CLIP instruction, 3-157
clobbered
defined, 2-55
registers, 1-195, 1-197
register sets, 1-198
clobber string
rules for, 1-109
specifiers, 1-109
clocale header, 3-36
clock (processor time) function, 3-44, 3-48, 3-115
CLOCKS_PER_SEC macro, 3-21, 3-44, 3-46
clock_t data type, 3-21, 3-44, 3-115
close function, 3-52
cmath header, 3-36
cmatmadd (complex matrix + matrix addition) function, 3-117
cmatmmlt (complex matrix * matrix multiplication) function, 3-119, 3-128
cmatmsub (complex matrix - matrix subtraction) function, 3-121
cmatsadd (complex matrix + scalar addition) function, 3-123
cmatsmlt (complex matrix * scalar multiplication) function, 3-124
cmatssub (complex matrix - scalar subtraction) function, 3-126
cmlt (complex multiply) function, 3-128
C++ mode compiler switches
-anach (enable C++ anachronisms), 1-71
-check-init-order, 1-72
-eh (enable exception handling), 1-73
-no-anach (disable C++ anachronisms), 1-75
-no-demangle (disable demangler), 1-75
-no-eh (disable exception handling), 1-75
-no-implicit-inclusion, 1-75
-no-rtti (disable run-time type identification), 1-75
-rtti (enable run-time type identification), 1-76
cmultr__conj_fr2x16 built-in function, 1-157
cmultr__conj_fr2x16_sat built-in function, 1-157
cmultr_fr2x16 built-in function, 1-157
cmultr_fr2x16_sat built-in function, 1-157
code optimization, enabling, 1-49
code sequences, 1-263
command-line interface, 1-5 to 1-76
comparison operators, int2x16 values, 1-135
compiler
C extensions, 1-88
code optimization, 2-2
errors, maximum, 1-67
optimizer, 2-4
prelinker, 1-85
switches, -always-inline, 1-26
switches, -never-inline, 1-43

compiler asm construct
construct template, 1-102
I/O operands, 1-112
optimization, 1-112
template, 1-102
template operands, 1-106
with multiple instructions, 1-111
complex
absolute value function, 3-97
addition function, 3-98
conjugate function, 3-129
division function, 3-101
exponential function, 3-103
header file, 3-32
matrix + matrix addition function, 3-117
matrix * matrix multiplication function, 3-119
matrix - matrix subtraction function, 3-121
matrix + scalar addition function, 3-123
matrix - scalar subtraction function, 3-126
multiply function, 3-128
number support, 1-287
subtraction function, 3-141
vector dot product function, 3-143
vector + scalar addition function, 3-144
vector * scalar multiplication function, 3-145
vector - scalar subtraction function, 3-146
vector + vector addition function, 3-147
vector * vector multiplication function, 3-148
vector - vector subtraction function, 3-149
complex.h header file, 3-23
complex matrix * scalar multiplication function, 3-124
compound statements
as macros, 1-249
within expressions, 1-123
compute block registers
ALU summation, 1-277
X MAC, 1-277
(x & y) general, 1-276
Y MAC, 1-277
conditional code, in loops, 2-34
conditional expressions, with missing operands, 1-240
conj (complex conjugate) function, 3-129
const
keyword, 2-38
pointers, 1-29
qualifier, 2-38
-const-read-write compiler switch, 1-29
-const-read-write flag, 2-38
constructors
int2x16 values, 1-132
int2x32 values, 1-140
int4x16 values, 1-136
constructs
flow control, 1-113
from polar coordinates (polar function), 3-262
operand, assembly, 1-105
reordering and optimization, 1-111
with input and output operands, 1-112
with multiple instructions, 1-110
-const-string switch, 1-29
continuation characters, 1-43, 1-47
controlling code inlining, 1-204
conv2d (2-d convolution) function, 3-132
convolution transformations, 3-24
convolve (convolution) function, 3-130
copysign (copysign) function, 3-134
cos (cosine) function, 3-135
cosh (hyperbolic cosine) function, 3-136
cot (cotangent) function, 3-137

count_ones (count one bits in word) function, 3-138
count_ticks() function, 1-230
count zero crossings function, 3-358
__cplusplus macro, 1-247
C++ programming examples
  complex support, 1-287
  fract support, 1-286
crosscoh (cross-coherence) function, 3-139
crosscorr (cross-correlation) function, 3-140
C run-time
  library files, 3-5
  library header files, 3-9
  model, 1-252
C++ run-time
  library, using linker with, 3-7
  library files, 3-5
  library with exception handling, 3-5
  support library files, 3-5
  support library with exception handling, 3-5
csetjmp header, 3-36
csignal header, 3-36
C startup files
  ts_hdr_.dlb, 3-6
C++ startup files
  ts_hdr_cpp_.dlb, 3-6
cstdarg header, 3-36
cstddef header, 3-36
cstdio header, 3-36
cstring header, 3-36
csub (complex subtraction) function, 3-141
ctime (convert calendar time into a string) function, 3-80, 3-142
ctype.h header file, 3-11
cvecdot (complex vector dot product) function, 3-143
cvecsadd (complex vector + scalar addition) function, 3-144
cvecsmlt (complex vector * scalar multiplication) function, 3-145
cvecssub (complex vector - scalar subtraction) function, 3-146
cvecvadd (complex vector + vector addition) function, 3-147
cvecvmlt (complex vector * vector multiplication) function, 3-148
cvecvsub (complex vector - vector subtraction) function, 3-149
cycle count
  cycle_count.h header file, 3-11
  interrupt dispatcher, 3-236
  register, 3-40, 3-41, 3-48
cycle_count.h header file, 3-11, 3-40
cycles.h header file, 3-12, 3-22, 3-41, 3-42
CYCLES_INIT(S) macro, 3-42
CYCLES_PRINT(S) macro, 3-42
CYCLES_RESET(S) macro, 3-42
CYCLES_START(S) macro, 3-42
CYCLES_STOP(S) macro, 3-42
cycle_t type, 3-40

D

DAB built-in functions
  16-bit (byte-addressing mode), 1-162
  16-bit (word-addressing mode), 1-162
  32-bit, 1-162
  circular buffer, 1-162
  described, 1-161
data
  alignment pragmas, 1-181
  alignment registers, 1-278
  buffers, quad-word-aligned, 2-18
  fetching with loads wider than 32 bits, 2-18
  field, 3-51
  objectaggregate, 1-79
  packing, 3-60
  word alignment, 2-18

data types
  128-bit quad-word, 1-115
  16-bit, 1-131
  32-bit, 1-77, 1-140
  64-bit, 1-77
  64-bit integer, 1-78, 1-114
  alignment, 1-79
  default bit sizes, 1-77
  floating-point, 1-78
  fract, 1-234
  integer, 1-78
  scalar, 2-13
__DATE__ macro, 1-247
daylight saving flag, 3-21
-DCLOCKS_PER_SEC= compile-time switch, 3-47
-D (define macro) compiler switch, 1-29, 1-65
-DDO_CYCLE_COUNTS compile-time switch, 3-42, 3-48
-DDO_CYCLE_COUNTS switch, 3-40
debugging
  -glite (lightweight) switch, 1-37
  information for header file, 1-30
  reducing size of information, 1-37
  source-level, 1-36
  -s (strip debugging information) switch, 1-59
-debug-types compiler switch, 1-30
declarations
  mixed with code, 1-243
decrements in expressions, 1-233
dedicated registers, 1-273, 1-281
default
  device, 3-58
  sections, 1-212
  target processor, 1-57
-default-branch-(np|p) compiler switch, 1-30
default_section pragma, 1-120
define
  interrupt handling, 3-235
  signal handling, 3-298
delete operator, with multiple heaps, 1-270
demangler, disabling, 1-75
deque header file, 3-37
DevEntry structure, 3-50
device
  default, 3-58
  driver, 3-50
  drivers, 3-50
  identifiers, 3-50
  pre-registering, 3-56
device.h header file, 3-12, 3-50
DeviceID field, 3-51
device_int.h header file, 3-12
devtab.c library source file, 3-56
diagnostics
  control pragma, 1-223
  types of, 2-5
difftime (difference between two calendar times) function, 3-150
div (division) function, 3-152
division, *see* div, ldiv functions
dm memory keyword, 1-88, 1-115
DOUBLE32 qualifier, 1-214
DOUBLE64 qualifier, 1-214
DOUBLEANY qualifier, 1-214
__DOUBLES_ARE_FLOATS__ macro, 1-32, 1-247
-double-size-32|64 switch, 1-31
-double-size-any switch, 1-31
double type formats, 1-31
-dryrun compiler switch, 1-32
-dry (verbose dry-run) compiler switch, 1-32

DSP header files
  complex.h, 3-23
  defined, 3-22
  filter.h header file, 3-24
  libsim.h, 3-25
  listed, 3-23
  matrix.h, 3-26
  stats.h, 3-27
  vector.h, 3-28
  window.h, 3-29
DSP run-time. library files, 3-5
dual memory support keywords (pm dm), 1-88
dynamic heap, 1-271

E

__ECC__ macro, 1-247
__EDG__ macro, 1-247
__EDG_VERSION__ macro, 1-247
-ED (run after preprocessing to file) compiler switch, 1-33
-EE (run after preprocessing) compiler switch, 1-33
-eh (enable exception handling) compiler switch, 1-73
elfar (archive library), 1-3, 1-27
embedded C++ library, header files, 3-32
embedded standard template library, 3-2
  header files, 3-36
__emuclk function, 3-154
emuclk (get simulator cycle count) function, 3-154
emulated arithmetic
  avoiding, 2-14
-enum-is-int compiler switch, 1-34
environment variables
  ADI_DSP, 1-81
  CCTS_IGNORE_ENV, 1-81
  CCTS_OPTIONS, 1-81
  listed, 1-80
  PATH, 1-80
  TEMP, 1-80
  TMP, 1-80
errno global variable, 3-31
errno.h header file, 3-13
error messages, control pragma, 1-223
escape character, 1-243
-E (stop after preprocessing) compiler switch, 1-33
exception
  handler, disabling, 1-75
  handler, enabling, 1-73
  handling, 1-22
  header file, 3-33
__EXCEPTIONS macro, 1-73
expected_false built-in function, 1-129, 2-28
expected_true built-in function, 1-129, 2-28
exp (exponential) function, 3-153
exponentiation, 3-75, 3-77
extensions, using compiler language, 1-4
external interrupts, 3-16
extractors
  2x16 from a 4x16 value, 1-137
  and expanders, int2x16 values, 1-133
  int2x32 values, 1-141
-extra-keywords (enable short-form keywords) compiler switch, 1-34
EZ-KIT Lite system, 3-12, 3-50, 3-59

F

fabs (float absolute value) functions, 3-155
false, *see* Boolean type support keywords (bool, true, false)

Fast Fourier Transforms (FFT), 3-24
fast N point complex input FFT (cfftf) function, 3-111
fast N point real input FFT(rfftf) function, 3-285
fast radix-2 algorithm, 3-24, 3-279
favg (mean of two values) functions, 3-156
fclip functions, 3-157
fclose function, 3-158
feof function, 3-159
ferror function, 3-160
fflush function, 3-172
FFT
  function versions, 3-24
  twiddle factors, 3-336
  twiddle factors for a fast FFT, 3-338
fgetc function, 3-173
fgetpos function, 3-174
fgets function, 3-176
file
  annotation position, 2-84
  attributes, 1-34, 1-293
  extension, 1-6, 1-8, 1-9
  searches, 1-8
-file-attr switch, 1-34
fileID field, 3-64
file I/O
  extending to new devices, 3-50
  support, 3-49
__FILE__ macro, 1-248
filename, -@ filename switch, 1-23
-@ filename (command file) compiler switch, 1-23
filter.h, 3-24
filters, signal processing, 3-24
finite impulse response filter., 3-161
fir_decima (FIR decimation filter) function, 3-163
FIR (finite impulse response filter) function, 3-161
fir_init macro, 3-168
fir_interp (FIR interpolation filter) function, 3-166
-flags (command line input) compiler switch, 1-35
flags field, 3-61
float.h header file, 3-13
floating-point
  arithmetics, 1-78
  divides, 1-36
  hexadecimal constants, 1-240
floor (floor) function, 3-178
flow control operations, 1-113
FLT_MAX macro, 3-13
FLT_MIN macro, 3-13
fmax functions, 3-179
fmin functions, 3-180
fmod (floating-point modulus) function, 3-181
fopen function, 3-182
fopen() function, 3-58
-force-circbuf (circular buffer) compiler switch, 1-35
-force-circbuf switch, 2-42
-fp-associative (floating-point associative operation) compiler switch, 1-35
-fp-div-lib compiler switch, 1-36
fprintf function, 3-19, 3-184
fputc function, 3-190
fputs function, 3-191
fract
  header file, 3-33
  support, 1-286
fractional
  arithmetic operators, 1-235
  data, 2-40
  (fixed-point) arithmetic, 1-234
  mixed-mode operations, 1-236
  values, 1-234
frame pointer, 1-254

fread function, 3-192
free function, 1-268
freopen function, 3-194
frexp (separate fraction and exponent) function, 3-196
fscanf function, 3-197
fseek function, 3-201
fsetpos function, 3-203
fstream header file, 3-33
fstream.h header file, 3-38
ftell function, 3-204
-full-dependency-inclusion switch, 1-73
-full-version (display versions) compiler switch, 1-36
full-word memory access, 2-15
functional header file, 3-37
function calls, avoiding loops in, 2-35
function inlining, 1-95, 2-23
functions, primitive I/O, 3-17
fwrite function, 3-205

G

GCC compatibility extensions, 1-237
GCC compatibility mode, 1-237
gen_bartlett (generate bartlett window) function, 3-207
gen_blackman (generate blackman window) function, 3-209
general optimization pragmas, 1-203
gen_gaussian (generate gaussian window) function, 3-210
gen_hamming (generate hamming window) function, 3-211
gen_hanning (generate hanning window) function, 3-212
gen_harris (gen_harris window) function, 3-213
gen_kaiser (generate kaiser window) function, 3-214
gen_rectangular (generate rectangular window) function, 3-216
gen_triangle (generate triangle window) function, 3-217
gen_vohann (generate von hann window) function, 3-219
getc function, 3-220
getchar function, 3-221
get_default_io_device, 3-57
get phase of a complex number (arg function), 3-79
gets function, 3-222
-g (generate debug information) compiler switch, 1-36
-glite switch, 1-37
global asm statements, 1-99
globvar global variable, 2-37
gmtime (convert calendar time into broken-down time as UTC) function, 3-224
gmtime function, 3-31, 3-80, 3-240
GNU C compiler, 1-237
__GROUPNAME__ macro, 1-64

H

hash_map header file, 3-37
hash_set header file, 3-37
header, stop point, 1-221
header file control pragmas, 1-221
header files
  C run-time library, 3-9
  embedded standard template library, 3-36
  library, 3-9
header files (C++ access to C facilities)
  (C++ access to C facilities)

cassert, 3-36
cctype, 3-36
cerrno, 3-36
cfloat, 3-36
climits, 3-36
clocale, 3-36
cmath, 3-36
csetjmp, 3-36
csignal, 3-36
cstdarg, 3-36
cstddef, 3-36
cstdio, 3-36
cstdlib, 3-36
cstring, 3-36
header files (C run-time library)
assert.h, 3-11
ctype.h, 3-11
errno.h, 3-13
float.h, 3-13
iso646.h, 3-14
limits.h, 3-14
locale.h, 3-15
math.h, 3-15
setjmp.h, 3-16
signal.h, 3-16
stdarg.h, 3-17
stddef.h, 3-17
stdio.h, 3-17
stdlib.h, 3-20
string.h, 3-20
header files (embedded C++ library)
complex, 3-32
exception, 3-33
fract, 3-33
fstream, 3-33
iomanip, 3-33
ios, 3-33
iosfwd, 3-34
iostream, 3-34
new, 3-34
header files (embedded C++ library) *(continued)*
ostream, 3-34
sstream, 3-34
stdexcept, 3-34
stream, 3-34
streambuf, 3-35
string, 3-35
strstream, 3-35
header files (embedded standard template library)
algorithm, 3-36
deque, 3-37
fstreams.h, 3-38
functional, 3-37
hash_map, 3-37
hash_set, 3-37
iomanip.h, 3-38
iterator, 3-37
list, 3-37
map, 3-37
memory, 3-37
new.h, 3-39
numeric, 3-37
ostream.h, 3-38
queue, 3-38
set, 3-38
stack, 3-38
utility, 3-38
vector, 3-38
header files (standard)
device.h, 3-12
device_int.h, 3-12
heap extension routines
heap_calloc, 1-267
heap_free, 1-267
heap_malloc, 1-267
heap_realloc, 1-267
listed, 1-267

heap functions
alternate, 1-268
alternative interface, 1-269
calloc, 1-266
free, 1-266
initializing, 1-268
malloc, 1-266
realloc, 1-266
standard, 1-266
standard interface, 1-268
using multiple, 1-266
heap identifier, 1-267
heap initialization, 1-271
heap_install function, 1-271
heap interface, with multiple heaps, 1-270
-help (command line help) compiler switch, 1-37
hexadecimal floating-point constants, 1-240
-HH (list headers and compile) compiler switch, 1-37
histogram function, 3-226
-H (list headers) compiler switch, 1-37
hoisting, 2-57
__HOSTNAME__ macro, 1-63, 1-64

I

iALU (j and k) general registers, 1-275
iALU (j & k) special registers:circular buffering –B (base) & L (length), 1-277
IDDE_ARGS macro, 1-264
ifft2d (NxN point 2-d inverse input FFT) function, 3-230
ifft (N point inverse FFT) function, 3-228
__IGNORE_IDLE_BUILTINS__ macro, 1-128
-ignore-std switch, 1-75
__IGNORE_SYSREG_BUILTINS__ macro, 1-128
iir (infinite impulse response filter) function, 3-232
iir_init macro, 3-233
Implicit inclusion, 1-73
implicit inclusion, 1-222
implicit pointer conversion, 1-39
-implicit-pointers compiler switch, 1-39
-include (include file) compiler switch, 1-40
increments in expressions, 1-233
indexed
array, 2-22
initializer support, 1-124
induction variables, 2-33
init function, 3-51
initialization
and reset of length and base registers, 1-164
disabling memory, 1-47
enabling memory, 1-42
function twidfft, 3-336
order, checking, 1-73
program, 3-24
successive elements, 1-125
variable, 1-243
with part-word address, 1-93
initiation interval, 2-92
inline
asm statements, 2-24
avoiding code, 2-45
keyword, 2-23, 2-45
inline assembly language support keyword, *see* compiler asm construct
inline control pragmas, 1-204
inline function support keyword, 1-95
inline keyword, defined, 1-95
inline qualifier, 1-96, 1-205

inlining
automatic, 2-23
file position, 2-84
function, 2-23
ignoring section directives, 1-99
inner loop, 2-33
instantiation, template functions, 1-219
instructions
ACS, 1-171
addition, 1-146
ALU miscellaneous, 1-149
bit manipulation, 1-153
composition, 1-158
conversion, 1-149
decomposition, 1-158
DESPREAD, 1-174
executed by communications logic unit (CLU), 1-164
floating-point operation, 1-157
generated by built-in functions, 1-145
MAX_ADD, 1-165
MAX_SUB, 1-165
memory allocation, 1-158
miscellaneous, 1-158
multiplier, 1-155
PERMUTE, 1-170
RECIPS, 1-144
RSQRTS, 1-145
shifter, 1-152
subtraction, 1-146
system register access, 1-160
TMAX, 1-165
TMAX_ADD, 1-165
TMAX_SUB, 1-165
XCORRS, 1-176
int2x32 data type, 1-140
integer data types, 1-78
interfacing C/C++ and assembly, *see* mixed C/C++ assembly programming
interpolation factor, 3-167
interprocedural analysis, 1-40, 1-84, 1-207, 2-12
interrupt
dispatcher, 1-185, 3-237, 3-299
dispatcher, cycle count, 3-236
function, 3-235
handler, 1-185
handler pragmas, 1-185
non-reentrant handler, 3-238
interruptf function, 3-235, 3-237
interruptfnr function, 3-238
interruptnr function, 3-238
interrupt_reentrant pragma, 1-185
interrupt-safe functions, 3-31
interrupts function, 3-235, 3-237
interruptsnr function, 3-238
intrinsic, *see* built-in functions, 1-126
inverse Fast Fourier Transform (IFFT), 3-228
I/O
extending to new devices, 3-50
primitives, 3-49, 3-50, 3-58
primitives, data packing, 3-60
primitives, data structure, 3-60
run-time library, 3-5
support for new devices, 3-50
iomanip.h header file, 3-33, 3-38
iosfwd header, 3-34
ios header file, 3-33
iostream.h header file, 3-34, 3-38
IPA, 1-84, 1-207
-ipa (interprocedural analysis) compiler switch, 1-40, 2-12
iso646.h (Boolean operator) header file, 3-14
-I (start include directory) compiler switch, 1-38, 1-48
istream.h header file, 3-34
iterator.h header file, 3-37

K

keywords (compiler), *see* VisualDSP++ compiler C/C++ language extensions

L

language extensions (compiler), *see* VisualDSP++ compiler C/C++ language extensions
LDBL_MAX value, 3-90
leaf procedure, 1-262
libc_.dlb library files, 3-5
libcpp_.dlb files, 3-5
libcpprt_.dlb library files, 3-5
libdsp_.dlb library files, 3-5
libio_.dlb library files, 3-5
library
  byte-addressing mode, 1-94
  C run-time, 3-5
  functions, listed, 3-65
  header files, working with, 3-9
  source code, working with, 3-8
  source file, devtab.c, 3-56
libsim.dlb library, 3-5, 3-19, 3-154
libsim.h header file, 3-25, 3-154
libsim print routines, 3-25
libx_.dlb library files, 3-5
lightweight debugging information, 1-37
limits.h header file, 3-14
line breaks, in string literals, 1-242
__LINE__ macro, 1-248
linking, pragmas for, 1-206
list header file, 3-37
live, 2-56
-L (library search directory) compiler switch, 1-40
-l (link library) compiler switch, 1-40, 1-48
local, variables and temporaries, 1-256
locale.h header file, 3-15
localtime (convert calendar time into broken-down time) function, 3-240
localtime function, 3-31, 3-80, 3-224
log10 (base 10 logarithm) function, 3-243
logical operators, bitwise, 1-134
log (natural logarithm) function, 3-242
long, latencies, 2-38
loop
  annotations, 2-93
  control variables, 2-36
  counters, 1-278
  cycle count, 2-81
  epilog, 2-57
  exit test, 2-36
  flattening, 2-89
  identification annotation, 2-79, 2-80
  invariant, 2-57
  iteration count, 2-49
  kernel, 2-56
  optimization, 1-187, 2-49
  parallel processing, 1-192
  prolog, 2-57
  resource usage, 2-81
  rotation, 2-59
  rotation by hand., 2-31
  scheduled, 2-92
  short, 2-29
  trip count, 2-35, 2-85
  unrolling, 2-30
  vectorization, 2-50, 2-62
loop-carried dependency, 2-30, 2-32
lvalue
  GCC generalized, 1-239
  generalized, 1-239

M

__MACHINE__ macro, 1-64

macros
compound statements as, 1-249
defining, 1-29
__GROUPNAME__, 1-64
__HOSTNAME__, 1-63, 1-64
IDDE_ARGS, 1-264
__MACHINE__, 1-64
preprocessor, 1-246
__read_ccnt, 1-160
__REALNAME__, 1-64
__RTTI, 1-76
__SIGNED_CHARS__, 1-66
__SYSTEM__, 1-64
__TS_BYTE_ADDRESS, 1-28, 1-91
undefining, 1-65
__USERNAME__, 1-64
variable argument, 1-241
writing preprocessor, 1-249
malloc function, 1-268
-map (generate a memory map) compiler switch, 1-42
map header file, 3-37
math functions
ceil, 3-113
floating-point, 3-15
math.h header file, 3-15
math intrinsics, 1-143
matinv (matrix inversion) function, 3-244
matmadd (real matrix matrix addition) function, 3-245
matmmlt (real matrix matrix multiplication) function, 3-246
matmsub (real matrix matrix subtraction) function, 3-248
matrix functions, 3-27
matrix.h header file, 3-26
matrix inversion, 3-244
matrix transpose, 3-335
matsadd (real matrix scalar addition) function, 3-249
matsmlt (real matrix scalar multiplication) function, 3-250
matssub (real matrix scalar subtraction) function, 3-251
maximum performance., 2-44
MAX instruction, 3-252
max (maximum) functions, 3-252
-MD (make and compile) compiler switch, 1-41
mean (mean) function, 3-253
-mem (enable memory initialization) compiler switch, 1-42
meminit_.dlb files, 3-6
MemInit utility, 1-42, 1-47
memory
bank pragmas, 1-226
data placement in, 2-25
header file, 3-37
initialization, 1-42, 3-6
initialization, disabling, 1-47
initialization, enabling, 1-42
meminit_ files, 3-6
support keywords (pm dm), 1-115
Microsoft C/C++ compiler, 1-128
minimum code size, 2-44
MIN instruction, 3-254
min (minimum) functions, 3-254
missing operands, in conditional expressions, 1-240
mixed C/C++ assembly
programming, 1-252
reference, 1-283
mixed C/C++ assembly naming conventions, 1-284
mixed C/C++ assembly programming
asm() constructs, 1-100, 1-102, 1-106, 1-111, 1-112
mixed-mode operations, fractional, 1-236
mktime (convert broken-down time into a calendar) function, 3-255

-M (make only) compiler switch, 1-41
-MM (generate make rules and compile) compiler switch, 1-41
modf (separate integral and fractional parts) function, 3-257
modulo scheduling
  annotations, 2-93
  defined, 2-92
modulo variable expansion unroll, 2-92
-Mo (processor output file) compiler switch, 1-42
-MQ compiler switch, 1-42
M_STRLEN_PROVIDED bit, 3-62
-Mt filename (output make rule) compiler switch, 1-42
mu_compress (μ-law compression) function, 3-258
mu_expand (μ-law expansion) function, 3-259
mult_fr1x32 functions, 1-155
mult_i1x32 functions, 1-156
mult_i2x16 functions, 1-156
mult_i4x16 functions, 1-155
multicore support, 1-207
multidimensional arrays, 1-123
multi-line asm() C program constructs, 1-111
-multiline compiler switch, 1-43
multiple heap support, 1-270
multiple instructions, constructs with, 1-110
multiple pointer types, declaring, 2-53
multr_fr1x32 functions, 1-155
mult_u2x16 functions, 1-155

N

namespace std, 1-75
nCompleted field, 3-64
nDesired field, 3-63
-never-inline compiler switch, 1-43
new devices
  I/O support, 3-50
  registering, 3-55
new header file, 3-34
new.h header file, 3-39
newline, in string literals, 1-43, 1-47
new operator, with multiple heaps, 1-270
-no-align-branch-lines compiler common switch, 1-43
-no-alttok (disable tokens) compiler common switch, 1-43
-no-anach (disable C++ anachronisms) compiler switch, 1-75
-no-annotate (disable assembly annotations) compiler common switch, 1-43
-no-annotate-loop-instr compiler common switch, 1-44
-no-auto-attrs switch, 1-44
-no-bss compiler common switch, 1-44
__NO_BUILTIN macro, 1-248
-no-builtin (no built-in functions) compiler switch, 1-45
__NO_BUILTIN preprocessor macro, 1-45
__NO_BYTE_ADDRESSING__ macro, 1-128
-no-circbuf (no circular buffer) compiler switch, 1-45
-no-const-strings compiler switch, 1-45
-no-def (disable definitions) compiler switch, 1-45
-no-demangle compiler switch, 1-75
-no-eh (disable exception handling) C++ mode compiler switch, 1-75
-no-extra-keywords (disable short-form keywords) compiler switch, 1-46
-no-fp-associative compiler switch, 1-46
-no-fp-minmax switch, 1-46
no implicit inclusion, 1-222

-no-implicit-inclusion C++ mode compiler switch, 1-75
NO_INIT qualifier, 1-214
-no-mem (disable memory initialization) compiler switch, 1-47
-no-multiline compiler switch, 1-47
non-constant initializer support (compiler), 1-124
non-unit stride, avoiding, 2-35
norm (normalization) functions, 3-260
-no-rtti (disable run-time type identification) C++ mode compiler switch, 1-76
-no-saturation (no faster operations) compiler switch, 1-47
-no-std-ass (disable standard assertions) compiler switch, 1-47
-no-std-def (disable standard definitions) compiler switch, 1-47
-no-std-inc (disable standard include search) compiler switch, 1-48
-no-std-lib (disable standard library search) compiler switch, 1-48
-nothreads (disable thread-safe build) compiler switch, 1-48
N point complex input FFT (cfft) function, 3-104
N point inverse FFT (ifft) function, 3-228
N point real input FFT (rfft) function, 3-278
numeric header file, 3-37
NxN point 2-D complex input FFT (cfft2d) function, 3-109
NxN point 2-d inverse input FFT (ifft2d) function, 3-230
NxN point 2-D real input FFT (rfft2d) function, 3-283

O

-Oa (automatic function inlining) compiler switch, 1-49
-O (enable optimization) compiler switch, 1-48, 1-49
-Og (optimize while preserving debugging information) compiler switch, 1-49
-o (output) compiler switch, 1-52
open function, 3-51
open() function, 3-58
operand constraints, 1-107
operators, comparison, 1-135
optimization
  asm() C program constructs, 1-112
  code size, 2-44
  compiler, 2-4
  compiler code, 2-4
  default, 1-82
  enabling, 1-49
  for code size, 2-44
  for speed, 2-44
  procedural, 1-82
  reporting progress, 1-57
  switches, 2-2, 2-54
  turning on, 1-85
optimization and debugging performance, 1-49
-Os (optimize for size) compiler switch, 1-50
ostream header file, 3-34
outer loop, 2-33
out-of-line copy, 1-98
-overlay, 1-52
overlay pragma, 1-203
overlays, 1-52
-Ov num (optimize for speed versus size) compiler switch, 1-50
-Ov (optimize for speed vs. size) compiler switch, 1-52

P

packed 16-bit integer support, 1-132
part-word, 1-93, 1-162
-path-install (installation location) compiler switch, 1-53
-path-output (non-temporary files location) compiler switch, 1-53
-path-temp (temporary files location) compiler switch, 1-53
-path-tool (tool location) compiler switch, 1-53
-pchdir (locate PCHRepository) compiler switch, 1-54
-pch (precompiled header) compiler switch, 1-54
-pedantic (ANSI standard warnings) compiler switch, 1-54
-pedantic-errors (ANSI C errors) compiler switch, 1-54
peeled iterations, 2-86
peeling amount, 2-86
perror, 3-261
.pgo file
  collecting PGO data, 1-83
  PGO process, 2-10
  -pgo-session switch, 1-55
  -pguide switch, 1-56
-pgo-session switch, 1-55
-pguide (profile-guided optimization) compiler switch, 1-56
pipeline viewer, 2-38
placement support keyword (segment), 1-88, 1-119
pm memory keyword, 1-88, 1-115
pointer
  arithmetic action on, 1-242
  byte-addressing mode, 1-92
  incrementing, 2-22
pointer class support keyword (restrict), 1-88, 1-121
pointer-induction variables, 1-188
polar (construct from polar coordinates) functions, 3-262
polyphase interpolation filter, 3-166
-P (omit line numbers) compiler switch, 1-52
pow (raise to a power) function, 3-263
-pplist (preprocessor listing) compiler switch, 1-56
-PP (omit line numbers and compile) compiler switch, 1-53
#pragma alignment_region, 1-183
#pragma alignment_region_end, 1-183
#pragma align num, 1-182, 2-18
#pragma all_aligned, 1-188, 2-51
#pragma alloc, 1-193
#pragma always_inline, 1-96, 1-205
#pragma bank_memory_kind, 1-230
#pragma bank_optimal_width, 1-232
#pragma bank_read_cycles, 1-231
#pragma bank_write_cycles, 1-231
#pragma can_instantiate instance, 1-220
#pragma code_bank, 1-227
#pragma const, 1-194, 2-46
#pragma core, 1-207
#pragma data_bank, 1-228
#pragma default_section, 1-212
#pragma diag, 1-223, 2-7
#pragma different_banks, 1-188, 2-53
#pragma do_not_instantiate instance, 1-220
#pragma hdrstop, 1-221
#pragma instantiate, 1-289
#pragma instantiate instance, 1-220
#pragma interrupt, 1-185
#pragma interrupt_reentrant, 1-185
#pragma linkage_name, 1-206
#pragma loop_count(min, max, modulo), 1-188, 2-49
#pragma loop_unroll N, 1-189

#pragma must_iterate(min, max, modulo), 1-189
#pragma no_alias, 1-191, 2-53
#pragma no_implicit_inclusion, 1-222
#pragma no_pch, 1-223
#pragma noreturn, 1-194
#pragma no_vectorization, 1-192, 2-50
#pragma once, 1-223
#pragma optimize_as_cmd_line, 1-204, 1-226
#pragma optimize_for_space, 1-203, 1-206, 1-225
#pragma optimize_for_speed, 1-204, 1-225
#pragma optimize_off, 1-203, 1-225
#pragma optimize_{off|for_speed|for_space}, 2-49
#pragma param_never_null, 1-216
#pragma pure, 1-194, 2-47
#pragma regs_clobbered, 1-195, 2-47
#pragma regs_clobbered_call, 1-199
#pragma result_alignment, 1-203
pragmas
  alignment_region, 1-183
  alignment_region_end, 1-183
  align num, 1-182, 1-188
pragmas
  alloc, 1-193
  always_inline, 1-205
  bank_memory_kind, 1-230
  bank_optimal_width, 1-232
  bank_read_cycles, 1-231
  bank_write_cycles, 1-231
  can_instantiate instance, 1-220
  code_bank, 1-227
  const, 1-194
  core, 1-207
  data alignment, 1-181
pragmas *(continued)*
  data_bank, 1-228
  default_section, 1-212
  defined, 1-179
  diag, 1-223
  different_banks, 1-188
  do_not_instantiate instance, 1-220
  function side-effect, 1-193
  hdrstop, 1-221
  header file control, 1-221
  instantiate instance, 1-220
  interrupt, 1-185
  linkage_name, 1-206
  linking, 1-206
  linking control, 1-206
  loop_count(min, max, modulo), 1-188
  loop optimization, 1-187, 2-49
  loop_unroll N, 1-189
  memory bank, 1-226
  no_alias, 1-191
  no_implicit_inclusion, 1-222
  no_pch, 1-223
  noreturn, 1-194
  no_vectorization, 1-192
  once, 1-223
  optimize_as_cmd_line, 1-206, 1-226
  optimize_for_space, 1-204, 1-206, 1-225
  optimize_for_speed, 1-204
  optimize_off, 1-203, 1-225
  param_never_null, 1-216
  pure, 1-194
  regs_clobbered_call, 1-199
  regs_clobbered string, 1-195
  result_alignment, 1-203
  section, 1-212
  separate_mem_segments, 1-204
  stack_bank, 1-229
  suppress_null_check, 1-217

pragmas *(continued)*
system_header, 1-223
template instantiation, 1-219
vector_for, 1-192
weak_entry, 1-215
#pragma section, 1-212
#pragma separate_mem_segments (var1, var2), 1-204
#pragma stack_bank, 1-229
#pragma suppress_null_check, 1-217
#pragma system_header, 1-223
#pragma vector_for, 1-192, 2-50
#pragma weak_entry, 1-215
precompiled header, 1-54
predefined macros
__ADSPTS__, 1-247
__ADSPTS101__, 1-247
__ADSPTS201__, 1-247
__ADSPTS202__, 1-247
__ADSPTS203__, 1-247
__ADSPTS20x__, 1-247
__ANALOG_EXTENSIONS__, 1-247
__cplusplus, 1-247
__DATE__, 1-247
described, 1-246
__DOUBLES_ARE_FLOATS__, 1-247
__ECC__, 1-247
__EDG__, 1-247
__EDG_VERSION__, 1-247
__FILE__, 1-248
__LINE__, 1-248
__NO_BUILTIN, 1-248
__SIGNED_CHARS__, 1-248
__SILICON_REVISION__, 1-248
__STDC__, 1-248
__STDC_VERSION__, 1-248
__TIME__, 1-248
__TS_BYTE_ADDRESS, 1-248
predefined macros *(continued)*
__VERSION__, 1-248
__VERSIONNUM__, 1-249
__WORKAROUNDS_ENABLED, 1-249
prelinker, 1-85
preprocessor macros, 1-246
PrimIO device, 3-56
_primio.h header file, 3-60
__primIO label, 3-59
primiolib.c source file, 3-57
primitive I/O functions, 3-17, 3-60
PRINT_CYCLES(STRING,T) macro, 3-40
printf
extending to new devices, 3-50, 3-56
functions, 3-18, 3-19, 3-26, 3-264
printf() function, 3-56
print routines, 3-25
procedural optimizations, 1-82
procedure
calls, 1-258, 1-261
returns, 1-262
processor
clock rate, 3-46
counts, measuring, 3-39
time, 3-115
-proc (target processor) compiler switch, 1-57
profile-guided optimization
command-line arguments, 1-264
defined, 2-8
described briefly, 1-82
enabling, 1-56
-Ov num switch, 2-45
-Ov switch, 1-52
-pgo-session switch, 1-55
program termination, 3-18
-progress-rep-func compiler switch, 1-57

-progress-rep-gen-opt compiler switch, 1-58
-progress-rep-mc-opt compiler switch, 1-58
progress reporting, 1-57
prototypes, calling, 1-273
putc function, 3-265
putchar function, 3-266
puts function, 3-267

Q

qsort (quicksort) function, 3-268
quad-word
  boundary, 2-18, 2-19
  data type, 1-115
quad-word-aligned address, 2-18
queue header file, 3-38

R

raise (raise a signal) function, 3-270
raise to a power function, 3-263
rand function, 3-31
rand (random number generator) function, 3-273
-R directory (add source directory) compiler switch, 1-58
-R- (disable source path) compiler switch, 1-59
__read_ccnt macro, 1-160
read function, 3-53
realloc function, 1-268
__REALNAME__ macro, 1-64
real vector dot product function, 3-343
real vector + scalar addition function, 3-344
real vector * scalar multiplication function, 3-345
real vector - scalar subtraction function, 3-346
real vector + vector addition function, 3-347
real vector * vector multiplication function, 3-348
real vector - vector subtraction function, 3-349
reciprocal square root (rsqrt) function, 3-290
reductions, 2-31
__regclass() construct, 1-117
__regclass(unit) keyword extension, 1-89
registers
  ADSP-TS101, 1-274
  ADSP-TS201, 1-274
  Bit FIFO temporaries, 1-278
  callee preserved, 1-273
  circular buffering –B (base) & L (length), 1-277
  classification, 1-273
  clobbered, 1-195
  compute block ALU summation, 1-277
  compute block X MAC, 1-277
  compute block (X & Y) general, 1-276
  compute block Y MAC, 1-277
  data alignment, 1-278
  enhanced communications, 1-278
  for asm() constructs, 1-106
  iALU (j and k) general, 1-275
  iALU (j & k) special registers, 1-277
  loop counter, 1-278
  qualifier, __regclass, 1-117
  return address, 1-278
  scratch, 1-273, 1-274
  static condition flags, 1-278
regs_clobbered string, 1-196
remarks
  control pragma, 1-223
  defined, 2-6
remove, 3-274
remove() function, 3-58
rename, 3-275
rename() function, 3-58

reordering asm() C program constructs, 1-112
restrict
  keyword, 2-37
  qualifier, 2-37
restricted pointer, 2-37
restrict keyword, 1-121
return address register, 1-278
return value, 1-260
rewind, 3-277
rfft2d (NxN point 2-D real input FFT) function, 3-283
rfftf (fast N point real input FFT) function, 3-285
rfftf_mag function, 3-287
rfftf magnitude, 3-287
rfft_mag function, 3-281
rfft magnitude, 3-281
rfft (N point real input FFT) function, 3-278
rms (root mean square) function, 3-289
-rtti (enable run-time type identification) C++ mode compiler switch, 1-76
__RTTI macro, 1-76
run-time
  C/C++ library, 3-5
  C library files, 3-5
  disabling type identification, 1-75
  enabling type identification, 1-76
RUNTIME_INIT qualifier, 1-214

S

-save-temps (save intermediate files) compiler switch, 1-59
scanf function, 3-291
scheduling, 2-56
scratch registers, 1-274, 1-281
search path
  for include files, 1-38
  for library files, 1-40
section
  elimination, 2-44
  qualifiers, 1-212
-section (data placement) compiler switch, 1-60
-section id (data placement) compiler switch
  controlling default names, 1-120
section() keyword, 1-89, 1-119
sections, placing symbols in, 1-212
seek function, 3-54
SEG_ARGV memory section, 1-264
segment, *see* placement support keyword (segment)
separate fraction and exponent function, 3-196
separate_mem_segments pragma, 1-204
setbuf function, 3-293
set_default_io_device, 3-57
set header file, 3-38
setjmp.h, 3-16
setvbuf function, 3-19, 3-295
-show (display command line) compiler switch, 1-61
sideways sum functions
  int2x16 values, 1-135
  (int4x16 values), 1-139
sig argument, 3-235, 3-298
signal (define signal handling) function, 3-298
signalf function, 3-298, 3-299
signalfnr function, 3-300
signal handling
  external interrupts, 3-16
  non-reentrant, 3-300
  signal.h header file, 3-16
signal.h header file, 3-16
signalnr function, 3-300
signals function, 3-298, 3-300

-signed-bitfield (make plain bitfields signed) compiler switch, 1-62
-signed-char (make char signed) compiler switch, 1-62
__SIGNED_CHARS__ macro, 1-62, 1-66, 1-248
sign (sign) function, 3-297
__SILICON_REVISION__ macro, 1-62, 1-248
simulator
  library support, 3-5
  services, 3-25
single case range, 1-242
sinh (hyperbolic sine) function, 3-303
sin (sine) function, 3-302
-si-revision (silicon revision) compiler switch, 1-61
sizeof operator, 1-91, 1-242
snprintf function, 3-304
software pipelining, 2-59, 2-62
source annotations, 2-7
source code, library, 3-8
spill function, 2-56
sprintf function, 3-19, 3-306
sqrt (square root) function, 3-308
srand function, 3-31
srand (random number seed) function, 3-309
sscanf function, 3-310
-S (stop after compilation) compiler switch, 1-59
sstream header file, 3-34
-s (strip debugging information) compiler switch, 1-59
stack
  header file, 3-38
  pointer, 1-253
stack frame
  free space, 1-256
  incoming arguments, 1-255
  linkage information, 1-255
  local variables/temporaries, 1-256
  outgoing arguments, 1-256
  outgoing linkage, 1-256
  overview, 1-253
stage count, 2-92
START_CYCLE_COUNT macro, 3-40
startup files
  C, 3-6
  C++, 3-6
statement expression, 1-237
static condition flags register, 1-278
statistical
  functions, 3-28
  profiling, 2-7
STAT registers, 1-160
stats.h header file, 3-28
stdarg.h header file, 3-17
stdargs (varargs) mechanism, 1-273
__STDC__ macro, 1-248
__STDC_VERSION__ macro, 1-248
stddef.h header file, 3-17
stderrfd function, 3-55
stdexcept header file, 3-34
stdinfd function, 3-55
stdio.h header file, 3-17, 3-49
stdlib.h header file, 3-20
std namespace, 1-75
stdoutfd function, 3-55
stdout output stream, 1-33
STOP_CYCLE_COUNT macro, 3-40
streambuf header file, 3-35
strftime (format a broken-down time) function, 3-312
stride argument, FFT function, 3-24

string
  header file, 3-35
  literals with line breaks, 1-242
string.h header file, 3-20
string literals
  defined, 1-29
  multiline, 1-43, 1-47
string-to-double conversion, 3-316
string-to-float conversion, 3-318
string-to-integer conversion, 3-320
string-to-long double conversion, 3-324
string-to-long integer conversion, 3-322
string-to-long long integer conversion, 3-326
string-to-unsigned long integer conversion, 3-328
string-to-unsigned long long integer conversion, 3-330
strstream header file, 3-35
strtod (convert string to double) function, 3-316
strtof (string to float) function, 3-318
strtoi (string to integer conversion) function, 3-320
strtok function, 3-31
strtold function, 3-324
strtold (string to long double conversion) function, 3-324
strtoll (string to long long integer) function, 3-326
strtol (string to long integer) function, 3-322
strtoull (string to unsigned long long integer) function, 3-330
strtoul (string to unsigned long integer conversion) function, 3-328
-structs-do-not-overlap compiler switch, 1-62
struct tm, 3-21
struct/union fields, unnamed, 1-245
sub-word data types, 2-15
switches
  -A (assert) compiler switch, 1-23
  -align-branch-lines, 1-25
  -annotate-loop-instr, 1-27
  -bss (placing data in bsz), 1-27
  -build-lib (build library), 1-27
  C/C++ mode selection, 1-22
  -C (comments), 1-27
  -c (compile only), 1-28
  -char-size-8|32, 1-28
  -char-size-any, 1-28
  compiler common, 1-23 to 1-70
    -alttok (alternative tokens), 1-25
    -glite (lightweight), 1-37
  -const-read-write, 1-29
  -debug-types, 1-30
  -default-branch-(np|p), 1-30
  -Dmacro (define macro), 1-29
  -double-size-32|64, 1-31
  -double-size-any, 1-31
  -dryrun (terse dry-run), 1-32
  -dry (verbose dry-run), 1-32
  -ED (run after preprocessing to file), 1-33
  -EE (run after preprocessing), 1-33
  -enum-is-int, 1-34
  -E (stop after preprocessing), 1-33
  -extra-keywords (enable short-form keywords), 1-34
  -@ filename (command file), 1-23
  -flags (command-line input), 1-35
  -force-circbuf, 1-35
  -fp-associative (floating-point associative operation), 1-35
  -fp-div-lib, 1-36
  -full-version (display versions), 1-36

switches *(continued)*
-g (generate debug information), 1-36
-h[elp] (command-line help), 1-37
-HH (list headers and compile), 1-37
-H (list headers), 1-37
-I directory (include search directory), 1-38
-implicit-pointers, 1-39
-include (include file), 1-40
-ipa (interprocedural analysis), 1-40
-I- (start include directory), 1-38
-L directory (library search directory), 1-40
-l (link library), 1-40
-map filename (generate a memory map), 1-42
-MD (generate make rules and compile), 1-41
-mem (enable memory initialization), 1-42
-M (generate make rules only), 1-41
-MM (generate make rules and compile), 1-41
-Mo (processor output file), 1-42
-MQ, 1-42
-Mt (output make rules), 1-42
-multiline, 1-43
-no-align-branch-lines, 1-43
-no-alttok, 1-43
-no-annotate (disable alternative tokens), 1-43
-no-annotate-loop-instr, 1-44
--no-bss, 1-44
-no-builtin (no built-in functions), 1-45
-no-circbuf (no circular buffer), 1-45
-no-const-strings, 1-45
-no-defs (disable defaults), 1-45
-no-extra-keywords (disable short-form keywords), 1-46
-no-fp-associative, 1-46
-no-mem (disable memory initialization), 1-47
-no-multiline, 1-47
-no-saturation (no faster operations), 1-47
-no-std-ass (disable standard assertions), 1-47
-no-std-def (disable standard macro definitions), 1-47
-no-std-inc (disable standard include search), 1-48
-no-std-lib (disable standard library search), 1-48
-nothreads (disable thread-safe build), 1-48
-Oa (automatic function inlining), 1-49
-O (enable optimizations), 1-48, 1-49
-o filename (output file), 1-52
-Og (optimize while preserving debugging information), 1-49
-Os (optimize for size), 1-50
-Ov (optimize for speed vs. size), 1-52
-path-install directory (installation location), 1-53
-path-output directory (non-temporary files location), 1-53
-path-temp directory (temporary files location), 1-53
-path (tool location), 1-53
-pchdir (locate PCHRepository), 1-54
-pch (precompiled header), 1-54
-pedantic (ANSI standard warnings), 1-54
-pedantic-errors (ANSI C errors), 1-54
-pguide (profile-guided optimization), 1-56
-P (omit line numbers), 1-52
-pplist file (preprocessor listing), 1-56
-PP (omit line numbers and compile), 1-53

-proc identifier (processor), 1-57
-progress-rep-func, 1-57
-progress-rep-gen-opt, 1-58
-progress-rep-mc-opt, 1-58
-R directory (add source directory), 1-58
-R- (disable source path), 1-59
-save-temps (save intermediate files), 1-59
-section (data placement), 1-60
-show (display command line), 1-61
-signed-bitfield (make plain bitfields signed), 1-62
-signed-char (make char signed), 1-62
-si-revision version (silicon revision), 1-61
sourcefile (sourcefile parameter), 1-23, 1-25
-S (stop after compilation), 1-59
-s (strip debugging information), 1-59
-syntax-only (just check syntax), 1-63
-syntax-only (system definitions), 1-63
-threads (enable thread-safe build), 1-64
-time (time the compiler), 1-65
-T (linker description file), 1-64
-Umacro (undefine macro), 1-65
-unsigned-bitfield, 1-65
-unsigned-char (make char unsigned), 1-66
-verbose (display command line), 1-66
-version (display version), 1-66
-v (version and verbose), 1-66
-warn-protos (warn if incomplete prototype), 1-68
-w (disable all warnings), 1-67
-Werror-limit (maximum switches), 1-67
-workaround (code generator workaround), 1-68
-Wremarks (enable diagnostic warnings), 1-67
-write-files (enable file redirection), 1-69
-write-opts (user options), 1-69
-Wterse (enable terse warnings), 1-68
-xref file (cross-reference list), 1-69
symbols, placing in sections, 1-212
-syntax-only (just check syntax) compiler switch, 1-63
-sysdef (system definitions) compiler switch, 1-63
sysreg.h header file, 2-41
__SYSTEM__ macro, 1-64

T

tanh (hyperbolic tangent) function, 3-333
tan (tangent) function, 3-332
template
asm() in C programs, 1-102
assembly construct, 1-102
class, 1-289
inclusion, control pragma, 1-222
instantiation pragmas, 1-219
instantiations, 1-289
support in C++, 1-289
thread-safe functions, asctime function, 3-31
-threads (enable thread-safe build) compiler switch, 1-64
time (calendar time) function, 3-334
time.h header file, 3-21, 3-44, 3-46, 3-48
__TIME__ macro, 1-248
timer interrupts, 3-16
-time (time the compiler) switch, 1-65
time_t type, 3-21, 3-334
time zones, 3-21
-T (linker description file) compiler switch, 1-64
transpm (real matrix transpose) function, 3-335

trip count
defined, 2-92
loop, 2-85
maximum, 2-92
minimum, 2-50, 2-92
trip modulo, defined, 2-92
true, *see* Boolean type support keywords (bool, true, false)
__TS_BYTE_ADDRESS macro, 1-28, 1-91, 1-95, 1-248
ts_exit_cpp_.dlb C++ exit routine files, 3-6
ts_exit_.dlb C exit routine files, 3-6
ts_hdr_cpp_.dlb start-up files for C++, 3-6
ts_hdr_.dlb C start-up files, 3-6
twiddle table, 3-24
twidfftf function, 3-338
twidfft (initialization) function, 3-336
two-dimensional FFT, 3-109, 3-283
type, cast, 1-242
typeof keyword, 1-238

U

unclobbered registers, 1-198
ungetc, 3-340
unnamed struct/union fields, 1-245
-unsigned-bitfield (make plain bitfields unsigned) compiler switch, 1-65
-unsigned-char (make char unsigned) compiler switch, 1-66
__USE_RAW_BUILTINS__ macro, 1-128
__USE_RAW_XCORRS__ macro, 1-128
__USERNAME__ macro, 1-64
utility header file, 3-38
-U (undefine macro) compiler switch, 1-30, 1-65

V

varargs routines, 1-260
variable
argument macros, 1-241
length array, 1-241
length array support, 1-122
var (variance) function, 3-342
vecdot (real vector dot product) function, 3-343
vecsadd (real vector + scalar addition) function, 3-344
vecsmlt (real vector * scalar multiplication) function, 3-345
vecssub (real vector - scalar subtraction) function, 3-346
vector functions, 3-28
vector header file, 3-38
vector.h header file, 3-28
vectorization
annotations, 2-90
avoiding, 2-50
defined, 2-62
factor, 2-86
loop, 2-50
unroll and jam, 2-87
vecvadd (real vector + vector addition) function, 3-347
vecvmlt (real vector * vector multiplication) function, 3-348
vecvsub (real vector - vector subtraction) function, 3-349
-verbose (display command line) compiler switch, 1-66
-version (display version) switch, 1-66
__VERSION__ macro, 1-248
__VERSIONNUM__ macro, 1-249
vfprintf function, 3-350
virtual function lookup tables, 1-60, 1-120

VisualDSP++
C/C++ language extensions, 1-87
IDDE, 1-4
running compiler, 1-3
simulator, 3-12, 3-50, 3-59
volatile and asm() C program constructs, 1-112
volatile declarations, 2-5
volatile keyword, 3-49
volatile register set, 1-199
vprintf function, 3-352
vsnprintf function, 3-354
vsprintf function, 3-356
vtbl section identifier, 1-120
-v (version and verbose) compiler switch, 1-66

W

warning messages
control pragma, 1-223
disabling, 2-5
#warning directive, 1-245
-Warn-protos (warn if incomplete prototype) compiler switch, 1-68
-w (disable all warnings) switch, 1-67
-Werror-limit (maximum compiler errors) compiler switch, 1-67
-Werror switch, 2-6
-Werror-warnings switch, 1-67
window generators, 3-29
window.h header file, 3-29
word-addressing mode
pointers, 1-161
selecting, 1-4, 1-28
_wordsize.h header file, 3-60
-workaround compiler switch, 1-68
__WORKAROUNDS_ENABLED macro, 1-249
-Wremarks (enable diagnostic warnings) compiler switch, 1-67
-Wremarks switch, 2-6
-write-files (enable driver I/O redirection) compiler switch, 1-69
write function, 3-52, 3-53
-write-opts (enable driver I/O pipe) compiler switch, 1-69
writes, array element, 2-33
writing preprocessor macros, 1-249
-Wsuppress switch, 2-5
-w switch, 2-5
-Wterse (enable terse warnings) compiler switch, 1-68
-Wwarn switch, 2-6

X

-xref (cross-reference list) compiler switch, 1-69
__XSTAT registers, 1-160

Y

__YSTAT registers, 1-160

Z

zero_cross (count zero crossings) function, 3-358
ZERO_INIT qualifier, 1-214
zero length arrays, 1-241